THE BETTER GUIDE TO THE BBC MICRO

AN ALTERNATIVE USER'S MANUAL

GEOFF BAINS

DORSET COUNTY LIBRARY
201741054 0

The Better Guide To The BBC Micro

Geoff Bains

DORSET COUNTY LIBRARY
WITHDRAWN STOCK
FOR SALE
30p

ZB

Zomba Books

DORSET COUNTY
1 JUN 1984
LIBRARY CH

First Published in Great Britain in 1984 by Zomba Books, Zomba House,
165—167 Willesden High Road, London NW10 2SG

© Geoff Bains, 1984

ISBN 0 946391 52 1

All rights reserved. No part of this book may be reproduced in any form or by any means, except for the inclusion of brief quotations in a review, without prior permission in writing from the publisher

Typeset by Hobson Street Studio Ltd, Cambridge
Printed and bound in Great Britain
at the Alden Press, Oxford
Cover Designed by John Sims

Production Services by Book Production Consultants, Cambridge

First Edition

Chapter One

Inside information

In this first chapter we are going to have a look at exactly what a microcomputer is. You do not need any detailed knowledge of the electronics of a microcomputer to operate it, but some idea of what is going on inside the box will help you to use it all the more efficiently.

Every microcomputer is based around a very versatile chip called a 'microprocessor'. All the real 'computing' tasks are carried out by the microprocessor. The microprocessor is the ruler of the microcomputer.

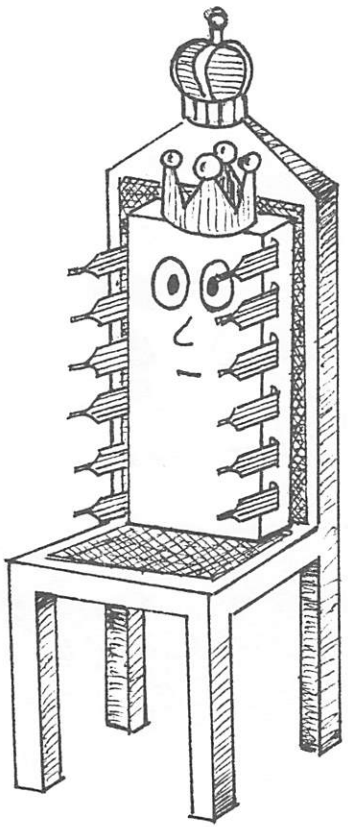


Fig. 1.1

Although microprocessors are clever by the standards of other chips, they are pretty dumb compared with us humans. All that a microprocessor can do is manipulate small numbers. It can add them, subtract them, and move them around. That's it. However what it can do, it does very quickly.

To do these things, the microprocessor must be given instructions. These are also in the form of numbers, and are stored in the second important part of the microcomputer: the memory.

The memory is made up of another sort of chip called a 'RAM' chip. There are usually several RAM chips inside any computer. Memory is rather like the pigeon holes that you see behind hotel reception desks to keep the residents' mail in. Each pigeon hole is labelled with the guest's room number. A RAM chip contains a lot of electronic pigeon holes or 'locations'. Each can contain a single number and each is labelled with an 'address'.

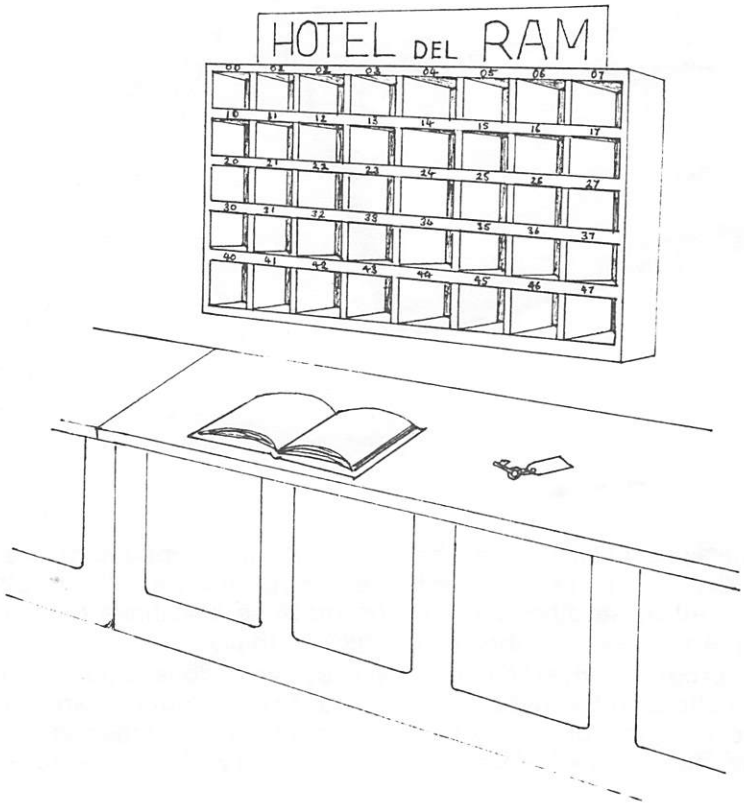


Fig. 1.2

To run a collection of instructions called a 'program', the microprocessor simply calls on these locations, one after another in sequence, using their addresses and carries out the instruction that it finds there. Rather like buying all the items on a shopping list as you read down it in a shop.

Fig. 1.3

That is about all that there has to be in a microcomputer. However the BBC micro, like all other home micros, also has a lot of other parts. All these other parts of the machine also have their own unique addresses for the microprocessor to use.

To program a microcomputer all we have to do is to put the right instructions in the right pigeon holes. These programs are called 'machine code' programs because we are using the number code understood by the fundamental machine that we have been looking at.

Machine code programming is not very nice at all. It certainly isn't a good idea to start with it. Machine code programs are just

collections of numbers. As you can see from the example they are not too easy to read and you can bet that they are not very easy to write.

```
AD    00    0D
8D    10    7C
AD    01    0D
8D    11    7C
60
```

Machine code program to print information onto the screen

Fig. 1.4

So that the microcomputer can be programmed by normal people who don't speak in numbers, there is another type of programming called 'high level language' programming.

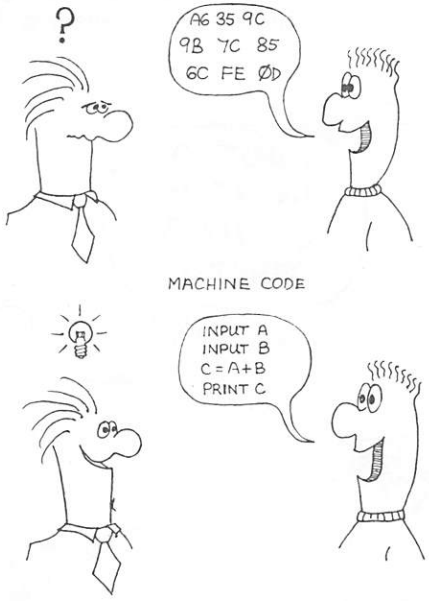


Fig. 1.5

The high level language that the BBC micro uses is 'BASIC'. BASIC stands for Beginner's All-purpose Scientific Instruction Code. It is just that, a code, or language, that beginners to computing can use to instruct their computer to perform all sorts of tasks, BASIC is not quite the Queen's English but it is a lot closer to it than machine code. It also makes the microprocessor appear to be smarter than it in fact is, by doing a lot of the work for you.

```
10  INPUT  S$
20  PRINT  S$
30  END
```

Basic program to print out a message onto the screen.

Fig. 1.6

The difficulty with BASIC is that the microprocessor doesn't understand a word of it. The microprocessor only understands machine code.

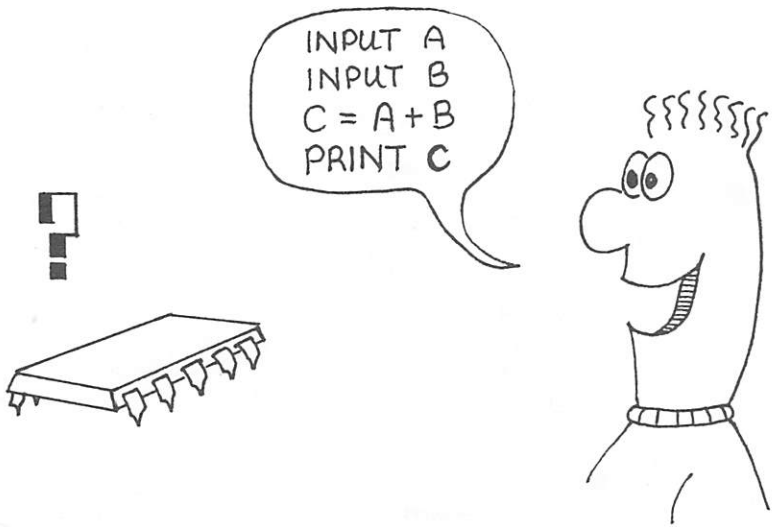


Fig. 1.7

To get around this problem the manufacturers of your computer include an 'interpreter' into the machine. The interpreter is another program, written by the manufacturers in machine code. The interpreter program takes your BASIC program and translates it into more machine code so that the microprocessor can understand it. The interpreter program is in a special sort of memory called 'ROM'.

ROM does not lose the information stored in it when you switch your micro's power off, and ROM cannot be changed by you (unlike RAM). That way, the interpreter program is always there whenever you need it.

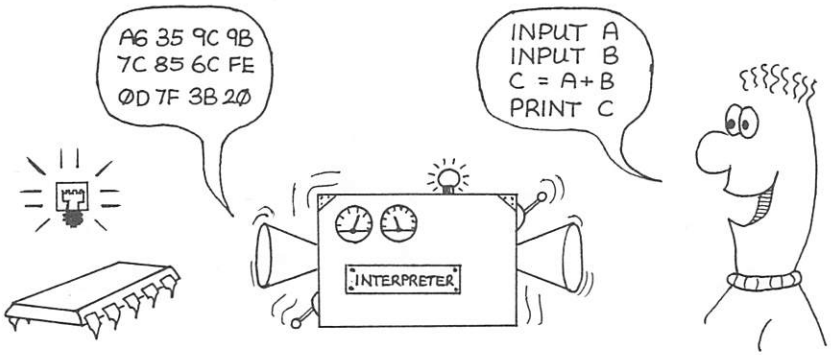


Fig. 1.8

Okay, so that's how a microcomputer will obey your BASIC program but how do you get the program in there in the first place? The answer is of course that you use the keyboard and the TV screen. Alongside the interpreter, there is another resident program in a ROM in your BBC. This is called the 'operating system', or OS. All the OS does is to look after your interests. It reads the keyboard switches to see if you're typing anything in at the keyboard, and puts your words up onto the screen. The operating system program is being used by the microprocessor all the time to do these administrative tasks.

It does all kinds of other things too. Like helping you edit your program, it looks after the casette recorder outlet, deals with all the graphics and sound, and so on. To help the OS program do these things there are lots of other chips inside your micro. There are video chips for the display, sound chips, and cassette driver chips.

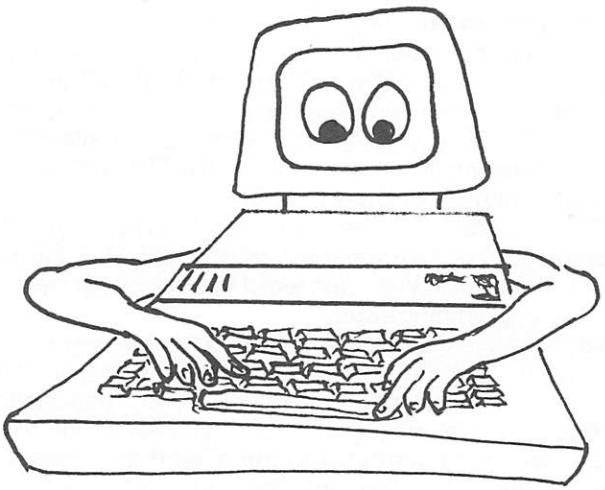
There's your BBC computer in a nutshell. When you type in a program, the resident OS program takes what you're typing and puts it up onto the screen so you can see it and edit it, and puts it into the micro's memory. When you run your program, the interpreter

program translates it into the machine code that the microprocessor understands and can act on and feeds it to the microprocessor. Any dealings with the keyboard, screen or sound in your program, and the microprocessor breaks off to use the OS to do all the dirty work, just like it did for your typing.

That's how the rudiments of your BBC work. If all this seems very complicated for rudiments, don't worry. Have another look at this chapter later. For now, all we have to do is to turn all this to our own advantage. We must learn to program your micro.

Chapter Two

Getting your fingers out



We have learnt that your BBC micro is only so much useless machinery without the series of instructions that we call a program. Now we shall have a look at how BASIC programs can be entered into your BBC: how to use the keyboard and screen. We shall assume that you have followed the first section of your BBC User Guide successfully and your TV screen is showing the words "BBC Computer 32K BASIC".

Your BBC is equipped with an excellent quality keyboard made up of a somewhat daunting seventy-four keys. Most of these are marked with letters and numbers. The rest have various symbols and words on them. The BBC micro's keyboard has been designed to be very easy and comfortable to use. If you have used an electric typewriter, and especially if you are proficient at typing, you will appreciate the quality of your micro's keyboard. If you are new to electronic keyboards then have a quick practise on it now. Just press any of the letter or number keys that you feel like. Get the feel of it.

You'll find that if you keep your finger down on a key, after about half a second, the micro will produce that letter at a rate of knots. When you remove your finger, it stops. This is known as auto repeat. Auto repeat helps when you're typing a large number of the same letter, but it also means that you can't leave your hands resting on the keys.

The letters that you press, of course, come up onto the screen. That's the operating system at work. All the time there is a flashing line just ahead of your typing. This is known as the cursor. It marks the position where the next letter to come up on the screen will appear, and informs you that the micro is expecting some typing from you. If you don't like a letter that you have typed press the key marked <DELETE> and the last letter will be removed.

The BBC will continue to take anything that you type in until you tell it that you have finished your instruction. It will assume that everything that you type in at this stage is an instruction and will act on it if it can understand you. You tell it that you have finished an instruction by pressing the key marked <RETURN> on the right hand edge of the keyboard. Go on, try it.

Your BBC has probably come back with the message 'Mistake'. This is because it did not understand what you were trying to tell it with your typing practice. We must learn to speak your micro's own versions of the English language.

The message, 'mistake', is known as an error message. This is not a report of an error that the micro has made, it tells you that it has noticed that you have made a mistake. Your BBC micro is not very clever. It cannot guess at what you mean if you do not give accurate instructions. If you had a written shopping list that included '3 lb of potetoes' you could guess that 'potatoes' were intended.

Your microcomputer cannot do this. When it comes across something that it doesn't understand it will tell you so with an error message. There are various messages that it can give. Which one appears depends on the circumstances and your error. You will find a list of the messages and the errors to which they refer on page 474 of the User Guide. Some of these may not make much sense to you now, but hopefully they will do as we go on.

Switch off your BBC and switch it on again to clear out all the rubbish you have been telling it. Now have a look at the bottom left part of the keyboard. There are three little lights there labelled 'cassette motor', 'caps lock' and 'shift lock'. Forget the cassette motor light for the minute. The caps lock light should be on. Press the key above it marked <CAPS LOCK> and you will see it go off. Press it again and it will come on again. Great, isn't it?

Now press the key marked <SHIFT LOCK>. The caps lock light has gone out and the shift lock light has come on. Play with these keys for a while and you'll see that only one can be on at a time or both can be off. That's very pretty, but what do they do?

Press these two keys untill both lights are off. Now type the keys marked 'ABCD1234'. You should get 'abcd1234' written out on the screen, just like a typewriter.

When the shift lock light is on, all of the keys on the keyboard give their shifted characters. If you know typewriters, you'll know that these are capital letters and the upper of the two symbols, for those keys with two symbols on them.

Press the <SHIFT LOCK> key to turn on its light and type in 'ABCD123'. Press <RETURN> after it if you want, if won't harm your micro, just baffle it a bit.

Low and behold it comes up on the screen as 'ABCD!"£$'. The key labelled <SHIFT> can be used (when the shift lock is not on) to produce this effect for as long as you hold it down. In fact there are two <SHIFT> keys, one each side, for convenience and for fast typing. Turn the shift lock off and type the above gibberish in again but this time hold down the <SHIFT> all the time you are doing it. You get the same result, right? The shift lock is there to save you holding down the <SHIFT> all the time.

Now turn on the caps lock with the <CAPS LOCK> key. Type in the letters again and you'll get 'ABCD1234'. When the caps lock is on the letter keys are shifted (and become capitals) but all the other keys are left alone. The caps lock makes all the letter keys give capital letters while the other keys give the lower symbol, or the upper one if they are pressed in conjunction with the shift key. This is very useful on the BBC micro because it will only understand commands given to it in capital letters. That is why your micro comes on with the caps lock in operation.

Okay, just one more: If you turn on the caps lock by pressing the <CAPS LOCK> key at the same time as the <SHIFT>, you will find that its effect is apparently the same. However if you now type a letter with the shift key again held down, the caps lock action is cancelled for as long as you hold down the shift key. In other words the caps lock light may as well be out when the <SHIFT> is held down. Try it.

That is the really confusing bit over. Try out all the possible combinations of the locks and <SHIFT> with a bit of random typing. Then if you're still not sure what is going on, read this section again and cast an occasional eye over this table

Lock	Letters	Others	Letters +Shift	Others +Shift
None	Small	Lower	Caps.	Upper
Shift Lock	Caps.	Upper	Caps	Upper
Caps. Lock	Caps.	Lower	Caps.	Upper
Shifted Caps. Lock	Caps.	Lower	Small	Upper

Fig. 2.1

Right, now we can get down to the nitty gritty of actually telling your micro to do something. Don't worry too much about exactly what we are telling it to do in this chapter, but try and learn the methods that we are using to instruct it. Remember: computers are not really very smart at all.

There are two ways in which a computer may be given instructions. The first is known as 'Immediate mode'. In immediate mode an instruction is typed in at the keyboard and <RETURN> is pressed. The computer will then obey that instruction immediately (if it can understand it). In the second mode the micro is told that the instruction is part of a program. This is done by putting a number (known as a line number) in front of the instruction. No instruction starts with a number so the micro can recognise a line number when you type one in. The micro will not act on an instruction given in this way when <RETURN> is pressed but will store the instruction and the line number in its memory. Only when the command to run the program is given will the micro obey the instruction. It will obey any number of instructions stored like this, in the order of their line numbers. First, though, lets take a quick look at immediate mode.

There are many instructions that can be given to your BBC micro in immediate mode. The most common of these is probably LOAD to transfer a program previously stored on cassette into the computers memory. However, a whole section has been devoted to storage, so we won't dally on that subject any further.

Your TV screen is most probably covered with the gibberish you were typing earlier, so let's look at ways to remove this.

The first method is to switch the computer off then on again. This is not really a command, nor is it necessary, but a similar effect can be obtained by pressing the <BREAK> key at the top right of the keyboard. Doing this at any time will make the computer break off from whatever it is doing at the time and assume a state very similar to when you first switch it on. Try it. Now try pressing <BREAK> along with the <CTRL> key (this is short for 'control') above the left <SHIFT>. You will see the micro doing just what it does when you first switch it on: it beeps and displays the now familiar message. <CTRL> <BREAK> resets the BBC micro to its switch on state. However these are pretty drastic methods of clearing the screen.

A simpler one is to type CLS. Don't forget the <RETURN> at the end. CLS stands for 'Clear Screen' and it does just that. Notice how the cursor is moved automatically to the top left of the screen (the 'home' position) ready for you to type in something on your clear screen.

Another method of clearing the screen is to change the screen 'mode'. As you have probably realised, the BBC micro can display information in different formats, with a greater or lesser amount on the screen and in more or fewer colours.

The mode that your BBC micro displays when switched on (or after a break) is number 7. That is what your looking at now. Try typing MODE 2 and <RETURN>. Changing mode clears the screen and homes the cursor. Type in something and you will see how characters in this mode are larger than in mode 7. Now try all the modes. One after the other, type MODE 0, MODE 1 and so on, up to 7. Be very careful to type '0' when you mean the number, and 'O' when you mean the letter. Your micro (being dumb) cannot guess what you really mean and you have to make it very clear. Some modes appear to be the same (0 and 3, 1 and 4, 2 and 5) but we will see the differences later in the book.

Okay, perhaps it is about time to try our hand at an actual program. We will start with a very simple program that will print out the names of your micro three times.

Before we give the first instruction we must tell the BBC, with a line number, that this is a program and not to be obeyed in immediate mode. The line number can be any whole number you like. As long as instructions that you want the computer to obey later

have larger line numbers you are all right. The first thing that we want the micro to do is to clear the screen so type in

```
10 MODE 4
```

Don't forget the <RETURN>. From now on you will have to remember the <RETURN> each time on your own. No more reminders!

As you can see, the screen has not cleared because you are now not giving immediate mode instructions, but writing a program. Congratulations! Type RUN followed by <RETURN> (okay, no more reminders from now!) and watch your program in action. This command starts the micro working on the program stored in its memory. You can RUN a program as many times as you like.

Not really very exciting, is it? That's because the only instruction in the program so far is one to clear the screen. The beauty of a program is that even when it has been run, it is still there ready to be run again and again, or changed or added to. That is what we are going to do now.

Type this into your BBC

```
20 PRINT "BBC MICRO"
```

The line number for this second instruction has to be bigger than the first one, for it to be obeyed second, PRINT is a command that will make the BBC print out whatever is inserted between a pair of inverted commas following it. Have a look on page 324 of the User Guide for details if you want. Now run your program again and you will see that now you get not only a clear screen but the name of your micro up there in lights. Our task was to print this name three times, so add this.

```
30 PRINT "BBC MICRO"
40 PRINT "BBC MICRO"
```

As this is all our program is to do we should add

```
50 END
```

This isn't strictly necessary in this case, but it is a good practice to get into. Now run the program and watch, you should be presented with

```
BBC MICRO
BBC MICRO
BBC MICRO
```

Very pretty. Let's assume that you don't think that this presentation is quite what you were looking for. You wanted a blank line between each name. We want to alter the program. First we must remind ourselves of how the program stands at the moment. Type in LIST. This immediate mode command gives you a list of the program lines in the micro's memory against the line numbers that you gave them. You should now see on the screen

```
10 MODE 4
20 PRINT "BBC MICRO"
30 PRINT "BBC MICRO"
40 PRINT "BBC MICRO"
50 END
```

To space the lines of print apart we must instruct the micro to print a blank line between each 'BBC MICRO'.

This can be done in two ways. Either by adding a ' to an existing PRINT command or by giving an extra PRINT command all on its own.

We'll try the second method first. To insert an instruction between existing ones all we have to do is give the new instruction a line number between the existing line numbers. Add a line 25 by typing

```
25 PRINT
```

The computer takes care of all the details (it's the interpreter and operating system again) and if you list out the program now you'll get

```
10 MODE 4
20 PRINT "BBC MICRO"
25 PRINT
30 PRINT "BBC MICRO"
40 PRINT "BBC MICRO"
50 END
```

Simple, isn't it? Any number of lines can be added in this way as long as you have left lots of line numbers free between the lines you have put in. That is why we spaced them ten apart. If we had labelled our instructions with line numbers 1 to 5, instead of 10 to 50, we would have had no room to fit in our extra line.

Run this new improved version and you'll find that the extra PRINT has done its job and inserted a blank line in this printout on the screen.

```
BBC MICRO

BBC MICRO
BBC MICRO
```

Now lets try the other method. We have got to change line 40 to read

```
40 PRINT ' "BBC MICRO"
```

We can do this in two ways. Firstly we can simply type in the new line in its entirety. Any program line that you type in with an existing line number, will replace the existing line. In this case the new line 40, with the apostrophe, will replace the old line 40, without one.

That's fair enough as far as it goes but it does mean you'll be doing a lot of typing if you make just a few little mistakes, and everybody does make mistakes. It's only really worth it for very short lines. What we want to do is just alter the line that is already in the memory. Fortunately the BBC micro is equipped with a very powerful editor to deal with just this situation.

The editing is done 'on screen'. That is: what you see on the screen can be altered. First therefore we have to put the line that we want to alter onto the screen.

We could just type LIST but this would put the entire program onto the screen and in the case of a long program this would both take a fair while and might scroll the line that we want off the top. Instead we type LIST 40. This will just put line number 40 onto your TV screen. Try it.

There are other forms of the LIST command too. Try typing LIST 10,30 and you'll find that the lines 10, 20, 25, and 30 are listed. LIST A,B will give you all the lines between and including numbers A and B. A must be a smaller number than B of course. Similarly LIST ,A will list out all the lines with numbers less than A, and LIST A, will give you all the lines greater than A. Have a go trying out these variations of LIST for yourself on your program. You can't do it any harm.

Okay — back to our editing. Put line 40 back onto the screen (you know how to, now!), and press the key marked with an upward pointing arrow. You'll see that the flashing cursor has turned into two. One that looks like the original has moved up one line on the screen and in its place has appeared a solid block. You are now in the editing mode. Pressing any of the cursor keys (that's the four with the arrows on), as you have done, will put the micro into editing mode.

Press the up cursor key again so that the small cursor moves up in line with the listed line 40. Now press the key marked <COPY>. Keep pressing it. See how the block cursor moves along in step with the small cursor and everything that the small cursor passes through is copied into the block cursor's position. When you've passed the end of the line 40. Press return. Now the cursors join

together again and the new line on the screen goes into the micro's memory. Of course all you have done so far is copy the line exactly as it was. The new line in memory is exactly the same as the old one — not too helpful. Check this with a LIST command if you want.

So let's now do a bit of real editing. Repeat this procedure again, but this time stop when the small cursor is under the first quote mark in the line 40. Now type in the extra bit of the line that we want. This is an apostrophe followed by a space. If you make a mistake while typing in the extra characters, pressing the <DELETE> key will remove the offending character as usual, but only from the new line. Now continue pressing the <COPY> key. The rest of the old line follows into the new line.

When you press <RETURN> at the end, all these characters go into the micro's memory along with the rest of the program. Do a LIST now and you'll see that line 40 is now just as we want it. Get the idea?

```
10 MODE 4
20 PRINT "BBC MICRO"
25 PRINT
30 PRINT "BBC MICRO"
40 PRINT ' "BBC MICRO"
50 END
```

There you have it: one edited program. RUN this program and you'll see the corrected print out as well

```
BBC MICRO

BBC MICRO

BBC MICRO
```

That's your first program. Not very useful but it served its purpose. Let's write another. First we must wipe out the old one. Type in NEW. This tells the micro that all that follows is a new program and is not to be added onto the old program. Another method of achieving this is to press <BREAK> or even <CTRL BREAK>. Just in case you do any of these when you didn't really mean to, you can still get your old program back by typing in OLD. However as soon as you start to enter your new program you can never regain the old one. So be careful! Try out all these methods and type LIST after each one. You'll find that the micro responds with nothing. Type in OLD and the micro remembers the program again and you can LIST the program after all.

Type NEW again and we are ready to enter your seond program. For your second program we shall really show your BBC micro

who's boss. We'll get it to put *your* name in lights. As a first line again type in

```
1 MODE 4
```

Now, much like last time, enter

```
2 PRINT "your name"
```

This time instead of typing the other print instructions in one by one we will use the editor to do all of the hard work for us. Use the cursor keys to move the small cursor to the beginning of the line you have just typed in. Now press the <COPY> key until the small cursor is under the 2 (the line number). Now press the right cursor key so that the small cursor skips over the 2. Type 3 and continue with the <COPY> key, pressing <RETURN> at the end. There, you have just created a line 3 exactly like line 2. However line 2 is still there. LIST the program to check. Now that's clever. The editor can be used in this way to copy anything that is written on the screen into a program line.

Do the same thing to create a line 4 and a line 5. If you want you can use a slightly different method that you may find easier (less moving around the keyboard). When you get to the bit when you have to skip over the line number, <COPY> it instead, then <DELETE> the copied figure from your new line and insert the new figure in the normal way. It makes no difference to the final result but some people find this second method easier.

Right. Your program should now look like this

```
1 MODE 4
2 PRINT "your name"
3 PRINT "your name"
4 PRINT "your name"
5 PRINT "your name"
```

Put in an END statement as line 6.

```
6 END
```

RUN this program. There's your name up on the (not so) big screen. No less than four times.

Now just suppose we were not satisfied with this printout either, just like last time. Suppose we want it to say how you are such a good programmer after your name, each time. There is more than one way we can alter the program to do this.

The first method is like one method that we used last time to alter your first program. We must insert lines between the existing ones to print the extra words. "Oh no," you cry. "We can't do that

because we haven't left any line numbers free in between the ones we've used. We have numbered our lines in steps of one". (You *are* crying this, aren't you?) The good news is that this doesn't matter quite as much as you thought.

All we have to do is renumber the lines. We could do this using the editor but this would take rather a long time. Instead we will use a very useful immediate mode command, RENUMBER. Type in this and then list the program. The micro has renumbered your program to start at line 10 and go on in steps of 10. You can make it renumber the program in different ways. Type RENUMBER 100,20. Now LIST it and you'll see that your program starts at line 100 and goes on in steps of 20.

Starting at line 10 with steps of 10 is what is known as the 'default' values for the command RENUMBER. These are the values that it will assume you mean unless you tell it different. We shall be meeting the idea of default values again. Your program should now look like this when listed

```
100 MODE 4
120 PRINT "your name"
140 PRINT "your name"
160 PRINT "your name"
180 PRINT "your name"
200 END
```

You now have plenty of room to add the extra lines we want. Type in a line 130

```
130 PRINT "IS AN EXPERT PROGRAMMER."
```

Use the editor to copy this into new lines 150, 170, and 190. Now your program should produce this when RUN.

```
your name
IS AN EXPERT PROGRAMMER

your name
IS AN EXPERT PROGRAMMER

your name
IS AN EXPERT PROGRAMMER

your name
IS AN EXPERT PROGRAMMER
```

The other method of adding to a program that we mentioned is to add instructions to an existing program line. So far we have used a separate line number for each instruction, but in fact any line number can relate to a number of instructions. These instructions must be separated by a colon (:) so that the micro knows where one instruction finishes and the next begins.

Delete your new line 190 by typing in the line number (190) and pressing <RETURN>. This replaces the existing line 190 with a new empty one that the micro ignores. Now use the editor to add the following onto the existing line 180

```
:PRINT "IS AN EXPERT PROGRAMMER"
```

Your line 180 should now read

```
 180 PRINT "your name":PRINT "IS AN EXPER
T PROGRAMMER"
```

This addition has exactly the same effect as the old line 190. As many commands as you may want can be put on one line in this way as long as the total number of characters in the line do not total more than 255. When you have filled up a program line in this way the computer will refuse to take any more characters for that line and will bleep at you if you try to type further characters in. Prove this to yourself by creating a line 500 filled with a lot of garbage. Type 500 then hold down the X key so that the auto repeat keeps on entering X's. When the total is reached no more X's will be accepted. Delete the line 500 when you're satisfied.

Well that's it really. You now know just about all there is to know about entering Basic programs into your BBC. There are just a few other points.

You may have noticed how the auto repeat also operates on the cursor keys. This can save a lot of effort when moving the small cursor all around the screen. A little tip here is that if the small cursor is moved off the top of the screen it will reappear at the bottom and vice versa. If it moves off the right of the screen it will reappear at the beginning of the next line on the left. If the line that you want to copy and correct is at the top of the screen, it is often quicker to press the down cursor key and approach it from above. Using the keys cleverly in this way can save you a lot of unneccessary cursor key pressing. With practice, you will soon get very proficient at operating the editing system and be able to do it without looking at the keyboard.

Pressing the <ESCAPE> key (top left of the keyboard) will abort an editing process and leave the line that you are editing unaltered. Pressing <ESCAPE> while a program is being listed will stop the

list. You can temporarily stop the listing of a program by pressing the <SHIFT> and the <CTRL> keys together. As soon as you release them the list will continue. Pressing <ESCAPE> while a program is running will stop that program. <ESCAPE> is a little like <BREAK>, only not so harsh.

A most useful command to help you write programs is AUTO. This is another immediate mode command. It saves you the bother of even typing in the line numbers. The micro can do this for you. Typing AUTO 100,20 will present you with the number 100 waiting for you to type in the instruction to be stored against that line number. As soon as you press <RETURN> at the end of the line, the next line number is produced. In this case it would be 120 because you set the step size at 20. Just like the RENUMBER command, AUTO has default values. If you only type AUTO, then the line numbers start at 10 with a step of 10.

There is one other use of the keyboard that you should know about now. Along the top of the keyboard there are ten red keys marked <f0> to <f9>. These are programmable keys. They can be made to do almost any action that you want. Programming them is very easy. We use the command, *KEY. This is an operating system command. You can tell by the asterisk that comes in front. Following this we write the number of the key that we want to program and then what we want it to do.

Supposing that we wanted key number five to produce the words 'Hello there' on the screen. (never mind why!). We can do this by typing in

```
*KEY 5 Hello there
```

If you now press the <f5> key you will see 'Hello there' on the screen. The red key (function key) acts as though you had typed in 'Hello there' from the keyboard. You can press <f5> as many times as you like. It will produce 'Hello there' every time. Of course if you now press <RETURN> the micro comes back with 'mistake' just as it would if you had typed this lot in. If we really wanted <f5> to put 'Hello there' on the screen we would have to do it like this

```
*KEY 5 PRINT "Hello there"
```

Anything, including all the Basic instructions can be programmed into the function keys. Try this last one and you will see that pressing <f5> puts the command up on the screen, but the micro doesn't obey it. This is because we haven't programmed a <RETURN> into the key. If you press <RETURN> after <f5> the words will be printed. If you want this to be a single key action, though, we have to include the <RETURN> into <f5>. We can't just press

<RETURN> at the point that we want it to appear because the micro will then take it as the end of our programming of the function key.

Instead we use the symbol, '!'. This is the upper symbol on the key next to the left cursor arrow. Unfortunately it appears differently in different display modes, but its action is always the same.

Incorporating '!' into an instruction is the same as typing the <CTRL> key. If the <CTRL> key is pressed along with some of the other keys, then instructions are given that never appear on the screen. For example <CTRL>M (that is the <CTRL> key and 'M' pressed at the same time) is the same as pressing <RETURN>. Try it. That is what we need to program into our <f5> key. We can now incorporate a <RETURN> into the key's instruction.

```
*KEY 5 PRINT "Hello there"!M
```

Now <f5> acts as though we had typed the instruction in and then pressed <RETURN>, so 'Hello there' is printed out on the screen.

Of course these keys can be used for less trivial tasks than silly messages. It is always handy to have a few of the keys set up to perform much repeated tasks when you are writing a program. Here is a selection that you may find useful to type in whenever you first switch on your micro.

```
*KEY 0 RUN!M
*KEY 8 LIST
*KEY 9 OLD!MMODE 4!MLIST!M
```

Now <f0> will run your program with a single touch. <f8> will list out your program. No !M has been added to this so that once pressed, you can add the desired line numbers and then press <RETURN>. <f9> has been programmed to be pressed after a <BREAK>. It restores the program with OLD then changes to mode 4 and lists it.

Any number of instructions can be put onto a single function key as long as the total number of characters that make up the instructions does not exceed 255. The function keys will remain programmed until you switch off the micro or press <CTRL><BREAK>. It is a good idea to make the first few lines of a program set up the function keys like we have done.

That really is it now. Before we move on, have a go at rewriting the last program so that it prints out four times that you are 'AN EXPERT PROGRAMMER BUT HAS A WAY TO GO YET' with the additional words on a new line. You can do this either by altering the present

program using the editor or by starting again and using the AUTO line numbering.

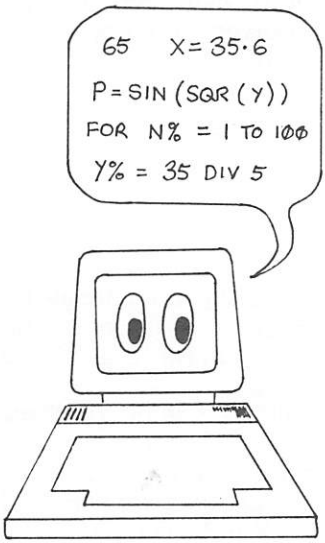
Even better, try both! If you get totally stuck, take a glance at the end of the chapter. You'll find out there what your program should look like.

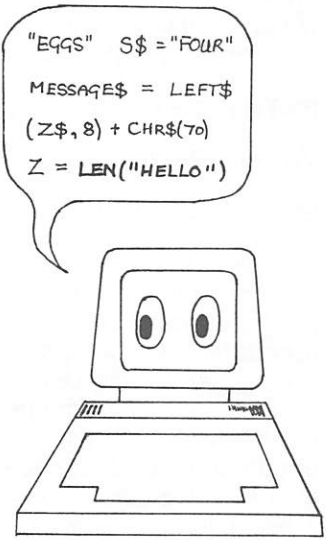
In the next chapter we are going to start to look at what you can put into your programs now that you know how to write them. Before you do that it is a good idea to know how to store your programs. If you want to leave off this book for a while in the middle of a chapter, you will want to know how you can save the program that you are working on. There is a whole section in the appendix to this book that tells you the ins and outs of using your cassette recorder with your micro. Have a look at at least the first few paragraphs of that now. You can read the rest at a later date.

```
 10 MODE 4
 20 PRINT "your name"
 30 PRINT "IS AN EXPERT PROGRAMMER"
 40 PRINT "BUT HAS A WAY TO GO YET !"
 50 PRINT "your name"
 60 PRINT "IS AN EXPERT PROGRAMMER"
 70 PRINT "BUT HAS A WAY TO GO YET !"
 80 PRINT "your name"
 90 PRINT "IS AN EXPERT PROGRAMMER"
100 PRINT "BUT HAS A WAY TO GO YET !"
110 PRINT "your name"
120 PRINT "IS AN EXPERT PROGRAMMER"
130 PRINT "BUT HAS A WAY TO GO YET !"
140 END
```

Chapter Three

Words and numbers





Now you have an idea of how to enter programs into your BBC micro, we must see what these programs can contain. In this chapter we will start to investigate this. At first, unfortunately, it is not going to be very rewarding. Until we have covered a few fundamental topics we can't go on to the really interesting bits. In this chapter we are going to look at numbers and words. A computer program is all about numbers and about words so this is very important (but not much fun!). Sorry.

Your micro can recognise numbers. Type in

```
PRINT 5+9
```

You have used PRINT as an immediate command here, so it is obeyed at once. The correct answer (14) is printed out. You can make your micro work out any sum in this way, using any of the normal mathematical functrions including those like SIN (sine), SQR (square root), and LOG. Note, however that for multiplication we use the sign, *, for division we use /, and for raising one number to the power of another we use ^.

There are a few functions tht you may not have met before. ABS(N), for example returns the absolute value of N. This is the equivalent positive number, ABS(–13.5) is 13.5 and ABS(6.9) is 6.9. SGN(N) will give you the sign of a number. It returns –1 for a negative number and +1 for a positive number. SGN returns zero if used on zero.

Brackets can be used in the normal way to force the micro to work out one particular bit of a long expression first. Without brackets it will calculate the answer in the order: powers, multiplication, division, addition, and then subtraction. In other words

```
PRINT 3+16*2
```

Will give 35, but

```
PRINT (3+16)*2
```

will give 38.

This order, known as operator precedence, can be found for all the maths operators on page 144 of the User Guide. A list of all the functions that you can use is included in the list of Basic keywords on page 483 of the User Guide.

Using just numbers in programs is not all that useful. What we need is a system where the values can change depending on the circumstances. Just such a system is used in computer programs. You can use what are known as variables. Variables are a way of storing, in your micro, numbers that change (vary!). Try this

```
X=15
```

You have just given (or 'assigned' in the jargon) the value of 15 to the variable called X. Your BBC will now associate the number 15 with the name, X, until you assign it a new value, or you press <BREAK>. Now type in these one after the other

```
PRINT X
PRINT 2*X
PRINT X^2
```

See how your micro behaves as though you had typed in the value 15 instead of X. Now assign a new value to X by typing

```
X=4
```

Type all of these calculations again. Hey presto! The correct answers for 4 are printed. A quick demonstration of the use of this ability your BBC has, would be the following program. Type it in and RUN it.

```
10 FOR N=1 TO 10
20   PRINT N,2*N,N^2
30   NEXT N
40 PRINT "FINISHED"
50 END
```

Line 10 and 30 together form what is known as a 'FOR-NEXT Loop'. The micro will repeat whatever is inside the loop (line 20 only, in this case) giving a successive value to the 'loop variable' each time, until the limit is reached.

The loop variable in this case is called N. We could have just as well called it X or anything else. The limit in this case is 10. So the variable, N is given a value, 1, then 2, then 3, and so on each time the program goes around the loop, until it reaches the value, 10. Then the program just keeps on going, without looping back to the start of the loop.

FOR-NEXT loops are a very easy way to repeat an operation a set number of times.

Never mind about the way the various answers are presented in the PRINT statement in line 20. This is to space out the answers neatly along the same line on the screen. We'll be dealing with this more fully in the next chapter.

Anyway, back to variables. Each time around the loop, the variable N was assigned a new value and several operations were carried out on N. Lots of different calculations were done, but the formula for the calculation was only written out once. This is what makes programming so useful.

We are now going to change our program to make it more useful and flexible. Change your program to look like this. You know how, now, don't you?

```
 5 INPUT START
10 FOR N=START TO START+10
20   PRINT N,2*N,N^2
30   NEXT N
40 PRINT "FINISHED"
50 END
```

When you RUN it you are presented with a question mark. This is all due to the new line 5. The command INPUT makes your micro come to you for a value for a variable in the middle of a program. If you type in a number when you are given the question mark, the program will assign this value to the variable name following the INPUT command. In this case, that variable name is START.

Variables can be called just about anything you want. Any combination of small and big letters can be used. You can use numbers too. This is handy for lists of similar variables which you could call VALUE1, VALUE2, VALUE3, and so on. However the name mustn't start with a number. It mustn't start with what is known as a 'Basic Keyword' either. These are just the commands that are used in BASIC. You can find a list of these on page 483 of the User Guide. One number that doesn't vary has a name already set out for it in your BBC micro. This is PI. When you are doing a calculation involving this constant you can call it by name to save you typing in 3.14159265 each time.

Line 10 takes the variable START and uses it for the start of the FOR-NEXT loop. The limit of the FOR-NEXT loop is now defined in line 10 as ten on from the value that you gave START.

Calling the variable by the name START is not necessary but it does mean that when you're reading through the program you can see straight away what the variables are being used for. It is good practice to call variables by useful names.

Okay, RUN the program and all will become a little clearer. When the question mark appears, type in a suitable number (say, 5) and watch how these variables have been put to use.

We can INPUT other numbers to control what values N is to take. Try altering your program to look like this

```
 5 INPUT START,FINISH,step
10 FOR N=START TO FINISH STEP step
20   PRINT N,2*N,N^2
30   NEXT N
40 PRINT "FINISHED"
50 END
```

Now we are not only determining the start value of N in the loop but also the finish and the step. Before we did not define the step that was added to N each time around the loop, so the default value of one was assumed (remember default values?).

With this program we can use any step size we want against the question mark when we RUN the program. Note that the names of the variables to be INPUT are separated by a comma in line 5. This is essential or the computer will think that we are talking about one variable called STARTFINISHstep. Note also that we have to call the variable 'step' and not STEP because STEP is a Basic Keyword (as used in line 10).

RUN this program. Give the program some sensible values for the three variables, say 15, 20, and 0.5. You can either give these one at a time with a <RETURN> after each, or all at once separated by a comma, and with a <RETURN> at the end. You can even try a negative value for step.

Okay, that's a brief introduction to what is known as 'real' variables. We are now going to look at 'integer' variables. The real variables can be assigned any value, whole number or decimal, between plus and minus two hundred million million million million million million, down to an incredibly fine accuracy. Integer variables on the other hand can only be given values that are whole numbers (intergers) between plus and minus a mere (!) two thousand million or so. Interger variables are named in the same way as real variables, but a percent sign (%) is put at the end. Any variable with a name ending in a percent sign, your BBC assumes is an interger variable.

Why then, you might wonder, do we need integer variables when real variables can do everything that integer ones can and more? The answer is that integer variables are faster. Because of the way the BBC micro stores numbers, the integer variables, although less versatile, can be retrieved from storage and dealt with quicker than real variables. Try this simple program. It not only demonstrates this but also introduces a new Basic Keyword, TIME.

Don't forget to type NEW first to wipe out the last program.

```
10 TIME=0
20 FOR I=1 TO 10000
30   NEXT I
40 PRINT TIME
50 END
```

TIME is a sort of variable. Inside your BBC there is a clock that constantly ticks (silently) away. The variable TIME is increased by one every 1/100th of a second to keep a running count of the elapsed time. Line 10 sets the variable TIME to zero so that we know where

we are starting. From here on TIME will keep a record of how long it is since line 10 was obeyed. Lines 20 and 30 form a simple FOR-NEXT loop with nothing inside it, just to give the micro 10,000 things to do. When the program has done that, it will print out how long it has taken (in hundredths of a second). RUN it and see.

Now change lines 20 and 30 to read

```
20   FOR I%=1 TO 10000
30     NEXT I%
```

Now the loop variable is an integer variable. Try the program again. See the difference? The integer version is tremendously quick. More than twice as fast. This can be very useful in applications where speed is very important, like animated graphics.

In other ways integer variables are pretty much like real variables and are treated in programs in the same way. Integer variables can of course only be assigned integer values. So X%=310/3 will assign the value 103 to X%.

Because of the difficulties that dividing integer variables can cause, there is a special form of integer division available on the BBC. This is peformed by the Basic Keyword, DIV. This gives the whole number part of the result of a division, only. So 15 DIV 2 is 7. To complement this, there is another command, MOD that will give the remainder of a division. 15 MOD 2 is 1.

There is a command that will convert a real number into an integer. X%=INT(10.25) will assign the value 10 to X%. INT gives the next whole number smaller than the real number it is operating on.

There is one other important difference between integer and real variables. The integer variables, A% to Z% are what is known as the resident variables. They maintain their values almost no matter what. Even RUN, NEW, and <BREAK> do not clear these variables unlike all others. This means that they can be used for passing values between different programs.

So far we have met two kinds of variables. You will be pleased to know that there is another. This is perhaps the most difficult for a new user to accept. We shall now have a look at 'string' variables.

In micro-speak words are known as strings because they are strings of characters. In fact any group of any characters is a string. String variables are distinguished by ending with a dollar sign ($). They can be assigned 'values' in the same way as the number type of variables. In this case, however, the 'value' is not a number value at all but a string of characters. Just as we did it before, try this

```
X$="Hello!"
```

The string of characters has to be contained inside inverted commas. The naming of string variables can be as versatile as the others, as long as the name ends with a $. Now type:

```
PRINT X$
```

See how the string variable acts just like the other, numeric, variables. Other operations can be carried out on string variables just as mathematical operations can be carried out on numeric variables. This is illustrated by the following short programs.

```
10 X$="HELLO"
20 Y$=" THERE"
30 Z$="!"
40 PRINT X$+Y$+Z$
50 END
```

Yes, words can be added together to make sentences. This addition of strings is the simple operation. RUN this program. It introduces a few very useful new commands.

```
 10 X$="The cat sat on top of the mat"
 20 PRINT X$
 30 PRINT
 40 L$=LEFT$(X$,7)
 50 PRINT L$
 60 PRINT
 70 R$=RIGHT$(X$,7)
 80 PRINT R$
 90 PRINT
100 M$=MID$(X$,8,15)
110 PRINT M$
120 PRINT
130 PRINT R$+M$+L$
140 END
```

Lines 40 and 70 produce two new strings from the original X$. These are the seven characters at the left hand end of X$ in the case of L$ and the right seven for R$. Needless to say any number of characters from one end can be turned into a new string. In this program we just chose seven. Line 100 created a new string out of the middle section of the original string. In this program we chose to take the 15 characters starting at character number 8 in the original string. Count along the characters in the original string in line 10 and you'll see how we got the string, " sat on top of ", for M$. Don't forget that spaces are characters as well.

Lastly, in line 130 we added and printed out our three new strings in the wrong order. As the positions of the strings in the original one had been carefully chosen, this new sentence still makes sense.

However, notice how the capital I that was at the beginning of the sentence now rather oddly appears in the middle.

Manipulating words in this way may not seem very useful in simple examples like this, but commands such as these are the basis for extremely useful programs. There are several special commands for manipulating string variables available on the BBC micro. VAL is a way of converting string variables to numeric variables. VAL(S$) will return the value of a string called S$ that consists of numbers, VAL("56.8") = 56.8. The string must start with plus, minus, or a number for the function to work, otherwise zero is returned. STR$ does the opposite. It will form a string that is made up of numbers. STR$(67.23) = "67.23". Another very useful function that connects string and numeric variables is EVAL.

EVAL will calculate the answer to a sum that resides in a string. EVAL("5*SQR(9)") = 15. This is very useful for entering mathematical equations into a program that is running. The equation can be entered as a string using INPUT and then calculated with EVAL.

There are still more things that we can do to words with the BBC micro

```
   10 INPUT X$
   20 L=LEN(X$)
   30 FOR N=1 TO L
   40   C$=MID$(X$,N,1)
   50   IF C$="e" OR C$="E" THEN X$=LEFT$(
X$,N-1)+"*"+RIGHT$(X$,L-N)
   60   NEXT N
   70 PRINT
   80 PRINT X$
   90 END
```

How about that then. The most complicated program you have written to date. RUN it and type in any sentence in response to the ? prompt from line 10. Your sentence is printed out again, but all the e's and E's have been converted to *'s! Perhaps we had better have a close look at this program because it uses a lot of new ideas.

The very first line has a slightly new idea in it. String variables can be inputted from the user (that's you) just like the numeric variables we were talking about earlier. Line 10 simply assigns whatever characters that you type in, up to a <RETURN>, to the string variable, X$.

Line 20 is a new one. The LEN keyword returns the number of characters (or length) of the string inside the brackets. You don't have to have a string variable name inside the brackets. The string itself, suitably enclosed in inverted commas, is just as good. Now we come to the difficult bit.

Lines 30 and 60 form a FOR-NEXT loop. In this loop the loop variable (it's called N again) is being given values from 1 to whatever the length of your sentence was.

Each time around the loop, the string variable, C$, is given a new contents. This is a single character of the original string, X$, that you typed in. MID$ is being used to give a single chartacter — that's why there is a '1' inside the brackets. This character is taken from a different place each time around the loop. It is the Nth character in the string. N, of course, is increasing each time around, so the overall effect is that C$ is assigned each character of your sentence in turn. Now what about the line 50?

Line 50 does the most important task of a computer program. It makes a decision. The general form of this Basic statement is very simple: IF something is so THEN do something. In this case the idea is: IF the character is an e THEN substitute a *. If the character is another letter then the program goes on to the next line and ignores the THEN part of the statement.

Your BBC can tell the difference between an e and an E so the condition that we are testing in line 50 must account for them both. It is fairly clear how this is done. The keyword, OR, is used. If the current contents of C$ is an e OR if the current contents of C$ is an E then we want to substitute the *. We shall be meeting this IF-THEN construction an awful lot more in our exploration of Basic.

Now how does the substitution of the * work? We have simply used these RIGHT$ and LEFT$ commands that we have already met. If there is found to be an e or an E at the Nth position then a new X$ is formed. This is made up of everyting in the old string left of the E and everything right of it along with an asterisk in the middle. Simple isn't it. Add these lines to your program which will show you how it works a bit better.

```
51 PRINT LEFT$(X$,N-1)
52 PRINT RIGHT$(X$,L-N)
```

These extra lines print out the bit before and after each character in the sentence as the program deals with it.

We could have equally well written this program in a totally different way. Like this for example. Try it.

```
   10 INPUT X$
   20 L=LEN(X$)
   30 N=INSTR(X$,"e")
   40 IF N<>0 THEN X$=LEFT$(X$,N-1)+"*"+RI
GHT$(X$,L-N):GOTO 30
   50 PRINT
   60 PRINT X$
   70 END
```

Yes that's right it's got several new Basic Keywords in it. Let's first look at the new keyword in line 30. INSTR is used to search one string for the appearance of another. We are using it here to search through your sentence, X$, for any e's that might be in it. This function returns a number representing the position of the first occurrence of the search string (e) in the main string (your sentence). This number is used in line 40, just like last time, for the LEFT$ and RIGHT$ functions to remove the e and substitute an asterisk.

The INSTR function will only look for the first occurrence of the search string. As your sentence may have several e's in it, this process of search and replacement must be repeated as often as needed. At the end of line 40, then, you'll see a new command, GOTO.

Whenever a program gets to a GOTO command it will jump to the line number following the GOTO command. In this case the line is number 30. So the program will go around and around this new type of loop between lines 30 and 40, removing all the e's from your sentence. Incidentally, GOTOs do not have to send the program back on itself to form a loop. They can also make the program skip a portion by giving a larger line number. In this case, however, a loop it is.

When a string is searched using INSTR but there is no occurence of the search string in it, INSTR gives a zero for the position. When the program gets a zero returned for N, then, all of the e's have been found.

We only want an asterisk to be substituted at the position, N, when N is not zero. That is why line 40 is an IF-THEN statement. The symbols, <> mean 'not equal to'. The symbols for all the operators like 'equal to', 'not equal to', 'greater than' and so on are given on page 145 of the User Guide. Only if there is an e left to replace will the program replace it and jump back to the start of the loop.

You may have noticed that this last program will only substitute asterisks for e's. Any E's in your sentence will be left well alone. This is because we only searched for e's in line 30. See if you can alter the program to search for E's. Then see if you can alter it to search for both.

There are many ways in which a program to peform a particular task can be written. We have seen just two here. Have another look at both of them. Try to understand how they work.

There is one last type of variable that we have not yet looked at. This is not so much a new sort of variable as a method of using those we have already met.

We have already seen how using numbers in a variable's name can make it easy to know what is going on when you're using a set

of similar variables. There is an even better method to use in some cases. Variables can be arranged in 'arrays'.

An array is simply a list of variables, all with the same name. The individual variables in the list are distinguished by a number suffix in brackets.

The best way to explain the idea of arrays is with a demonstration. So here goes. Try this program. It is a sort of computerised version of the old game of 'Consequences'.

```
   10 DIM N$(9),S(9),A$(9)
   20 PRINT "Input ten names followed by 1
"
   25 PRINT "for a male or 0 for a female.
"
   30 FOR I%=0 TO 9
   40   INPUT N$(I%),S(I%)
   50   NEXT I%
   60 PRINT "Now ten actions in the past"
   65 PRINT "tense. eg 'sat on', 'kissed'.
"
   70 FOR I%=0 TO 9
   80   INPUT A$(I%)
   90   NEXT I%
  100 M$=" met ":A$=" and ":SP$=" "
  110 REPEAT
  120   PRINT
  130   PRINT
  140   N1=RND(10)-1:N2=RND(10)-1:N3=RND(1
0)-1
  150   IF S(N2)=1 THEN S$="him" ELSE S$="
her"
  160   PRINT N$(N1)+M$+N$(N2)+A$+A$(N3)+S
P$+S$
  170   T=TIME+100
  180   REPEAT UNTIL TIME>T
  190   UNTIL FALSE
  200 END
```

Your longest program yet, and lots of new stuff too. The first line is one to remember. If you are ever using arrays in a program you must say so at the beginning of the program. DIM is short for dimension. This statement tells the computer how many variables there are in each of your arrays. In this program we are using three arrays. Two are string variable arrays, called N$ and A$, and one is a numeric variable array called S.

In the DIM statement each array name is followed by brackets containing the suffix of the last value in the list. All three arrays in this program have ten entries, but the first element is always labelled zero and not one, so the last element in this program is labelled nine.

Lines 20 to 50 simply store ten strings that you type in, into the array, N$. A FOR-NEXT loop increments the value of the loop variable, I%, and each time around a string is read in from the keyboard and assigned to the element in the array, N$, labelled with the suffix equal to I%.

At the same time as this is going on a similar thing is happening with the numeric array, S. After the names and sexes have been entered, exactly the same idea is used to enter the actions into a string array called A$.

By the time the program gets to line 100, there are three lists, or arrays, filled with names, sexes, and actions. Now we must use them for our game.

Line 130 contains a new keyword, RND. This is short for 'random'. RND returns a whole number, randomly chosen, between 1 and the number in the brackets. We are using this function here to randomly pick out names and actions. If the names and actions were not in arrays this would be very hard indeed. As we want numbers in the range zero to nine, we use the RND function to produce numbers in the range one to ten and subtract one from them. Three random numbers are produced and assigned to the variables, N1, N2, and N3.

These last variables are not part of an array. They just have numbers in their names to help make the program easier to read. We could have called them Tom, Dick, and Harry if we so wished. To be array elements they *must* have the number inside brackets and the array *must* be dimensioned with a DIM at the start.

Now line 150 just prints out the sentence made up of our randomly selected action and names. The other words for the sentence have been assigned to suitable string variables in line 100.

Notice how the word 'and' has been put into a string variable called A$. The computer will not confuse this with the array called A$ because the array has a number suffix in brackets after it. Arrays and single variables can have the same name without any confusion (in the micro's mind at least!).

Line 150 is an IF-THEN construction which decides whether to put a 'him' or a 'her' at the end of the sentence. It does this by looking in the array, S to see whether there is a one or a zero in the element corresponding to the randomly chosen second name. Using arrays in this way to cross refer between lists is a very important use. We shall use more of this in the chapter on working with data later.

The IF-THEN structure is a bit different there than the one we have met before. Now it is an IF-THEN-ELSE structure. If the condition is met (if N2 does equal 1 in this case) the THEN part of the structure is carried out in the normal way and the program continues onto the next line. If the condition is not met, however, rather than just

skipping onto the next line, the program will peform whatever instructions are in the ELSE part of the statement. This is a way to choose between two courses of action. The ELSE part can be added on to the IF-THEN construction or not as you need.

That's how our random sentences are created. There are just a couple of other points in this program. You'll see the keywords REPEAT and UNTIL in two places in this program. These form REPEAT-UNTIL loops, a very useful programming structure, not to be found very much outside BBC Basic.

REPEAT-UNTIL loops are very similar to FOR-NEXT loops. The difference is that we don't know in advance just how many times the program is going to go around the loop. Lines 110 to 190 form a REPEAT-UNTIL loop that repeats the part of the program that forms the random sentences. It will do this until the condition following the UNTIL is met. In this case the condition is the word FALSE. This is a little confusing. Actually what this results in, is to continue repeating the section of program (repeating the production of random sentences) until you press the <ESCAPE> key.

The idea of REPEAT-UNTIL loops is shown better in line 180. In the previous line a variable, T had been set to 100 on from the value of the TIME clock (remember TIME?). Line 180 just repeats itself again and again until TIME suppasses this value. It will do this after one second (one hundred centiseconds — the units of TIME). Line 180, then, gives the program a one second delay between each sentence.

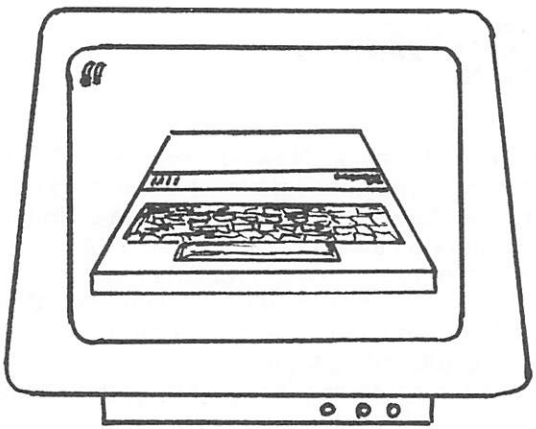
These two REPEAT-UNTIL loops are 'nested'. That is, one is inside another like a nest of tables. This means that every time around the outer loop the inner loop repeats as many times as it will. This is quite okay, but you must be careful not to cross the loops. The inner one must start and finish totally inside the outer. As long as this is done, the loops can be nested for as many levels as you want.

In this chapter we have covered a lot of ground. A lot of it is very confusing and not immediately obvious in its use. However, rest assured that we have not been wasting our time.

In the next chapter we are going to take a look at the screen display, but first have another look at the programs that we have written in this chapter. Experiment with them. Put in different values and see what happens. Learn how they work, and you'll be writing different ones of your own in no time.

Chapter Four

Screening processes



In this chapter we are going to take a closer look at the screen display of the BBC micro. The screen is the micro's most important method of communication with us. It is therefore very important that we learn to use it properly and fully.

So far in our programs, whenever we have wanted some text to appear on the screen, we have used a simple PRINT statement. This has resulted in each separate item to be printed appearing on a new line, up against the left hand side of the screen.

RUN the following program to remind yourself of this. If you have not just switched on your micro then type in mode 7, first.

```
10 FOR N=0 TO 20
20   PRINT N
30   PRINT 2*N
40   PRINT N^2
50   NEXT N
60 END
```

This program simply creates a lot of numbers. It uses one of those FOR-NEXT loops again, with a loop variable called N. Each time around the loop, 21 times in all, N is printed out, along with its double and its square.

When you RUN this program you will see all the numbers flash past as the display scrolls up the screen. You can halt it of course, by pressing both the <CTRL> and <SHIFT> keys, but even then you'll soon lose track of exactly which number is a 2*N and which an N squared. This is because all the numbers are on separate rows. What we want is to display the results in a table of some sort.

Change the program to look like this using the editing and RENUMBER.

```
10 FOR N=0 TO 20
20   PRINT N;2*N;N^2
30   NEXT N
40 END
```

You'll see the difference. Now all three PRINT statements are combined into one with semicolons (;) separating each result. The semicolon forces the next item to be printed to follow directly on from the last, with no new line or even a space.

We have a sort of table now but reading each line is a bit difficult because each result runs into the next. However things could be worse, try adding a semicolon onto the end of line 20 as well. Now we have all 63 results trying to fit onto the line one after the other.

What we need is some spaces to split up the printed results on each line

```
10 FOR N=0 TO 20
20   PRINT N;"    ";2*N;"    ";N^2
30   NEXT N
40 END
```

The spaces are contained in inverted commas so the PRINT statement puts them on the screen as they are. There is even a special command to save you typing out all these spaces. SPC(3) will print out three spaces.

```
10 FOR N=0 TO 20
20   PRINT N;SPC(3);2*N;SPC(3);N^2
30   NEXT N
40 END
```

SPC will print any number of spaces, but it is only usually worth using it if you want a lot. Just three, as we're using here, doesn't really warrant anything but spaces themselves.

These methods work but not very well. The columns of numbers do not line up. We need a tabulating function, such as you'll find on the better typewriters, to create tidy tables from figures like these. Not very surprisingly there is just such a function. Replace the semicolons with commas like this

```
10 FOR N=0 TO 20
20   PRINT N,2*N,N^2
30   NEXT N
40 END
```

Now run the program and you'll see a very nicely presented table of figures.

The screen is split into what are known as 'fields'. These are usually ten characters wide (though this can be changed by the more experienced user). When the BBC comes across a comma in a print statement it will print out numbers so they appear up against the right hand edge of the field. Words, however are printed to start in the next available field. Try this program and you'll see how this works.

```
   10 CLS
   20 PRINT 1,2,30,400
   30 PRINT "one","two","thirty","4 hundre
d"
   40 END
```

Using the comma in this way makes any form of tabulation easy. Lining up figures and their names becomes a cinch.

```
10 CLS
20 PRINT 3,"Number1"
30 PRINT 23,"Number2"
40 PRINT 123,"Number3"
50 END
```

The lining up is fine, but the numbers run into their names. This problem is easily avoided by using a semicolon at the start of the PRINT statement. This causes the first item in each statement to be printed directly following on from the start of the line, ie over to the left.

```
10 CLS
20 PRINT; 3,"Number1"
30 PRINT; 23,"Number2"
40 PRINT; 123,"Number3"
50 END
```

There is another way of positioning text or numbers on the screen. This uses the TAB command. RUN this program

```
10 CLS
20 PRINT 1
30 PRINT 2
40 PRINT 3'4
50 PRINT 5;6;7
60 PRINT 8,9,10
70 PRINT TAB(13,15);11 TAB(20) 12
80 END
```

This program demonstrates all the methods to position characters on the screen. 1 and 2 are printed using PRINT alone and so they appear at the left hand end of separate lines. 3, 4 and 5 also appear like this because they are separated in the PRINT statement by an apostrophe (') which, as we saw in the second chapter, is equivalent to typing in a second PRINT statement. 6, 7, and 8 are separated in their PRINT statement by semicolons, so they appear as 678.

The figures, 8, 9, and 10 are separated by commas so they appear neatly spaced at ten character intervals. The figure 11 however is in a most peculiar position. In fact it is exactly 13 characters positions from the right hand edge of the screen and 20 characters positions down from the top. Count them. These are the numbers that follow the first TAB command in line 70.

The TAB uses a co-ordinate system. TAB(x,y) will cause the cursor to move to the position, (x,y), according to the scheme shown below. The following semicolon causes the number, word, or whatever, to be printed to start at that position.

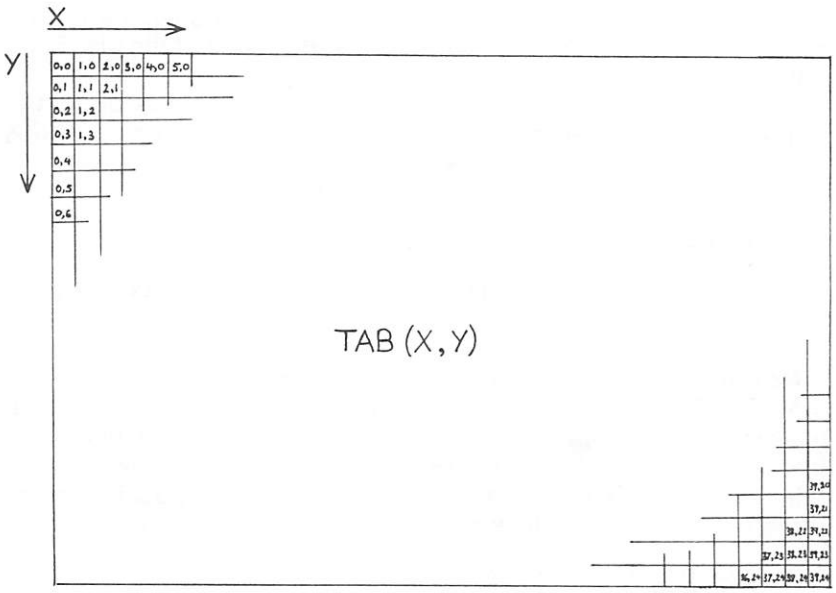


Fig. 4.1

The TAB command in line 40 has only one co-ordinate. The computer assumes a single co-ordinate to be the x co-ordinate. This shifts the cursor along the current line this number of positions. The 12, therefore, is 20 characters positions further along the same line as the 11.

TAB is very useful if you want to place characters on the screen at a higher position than you have already placed text so far. With the other PRINT statements, of course, the cursor will only move down the screen.

TAB is not the only method of moving the cursor around the screen. We can also use the keyword, CHR$. Try this

```
10 CLS
20 PRINT "X";"X"
30 PRINT "X";CHR$(10);"X"
40 END
```

Line 20 uses a perfectly normal PRINT statement to put two X's onto the screen next to one another. The semicolon prints the second X straight after the first as we have already seen. Line 30, however, contains a mysterious CHR$(10) in between the two X's and you can see when you RUN this program that the second X is put on the next line down from the first. It isn't at the beginning of

the line, though, as you'd expect if we had used a separate print statement, but directly below where it would appear if the CHR$ had not been there.

CHR$(10) moves the cursor down one line. Because all text is printed on the screen at the position of the cursor, the second X appears on the next line.

Try the program again with a new line 30

```
   10 CLS
   20 PRINT "X";"X"
   30 PRINT "X";CHR$(10);CHR$(9);CHR$(9);"
X"
   40 END
```

Now the second X appears on the next line and two positions to the right. CHR$ can be used in this way to maniplate the cursor, and thereby manipulate the position of text to be printed. Unlike TAB that we have already seen, CHR$ moves the cursor to a new position relative to its current position and not to an absolute position on the screen. There are in fact six cursor-moving CHR$'s. Here they are

CHR$(8)	Back one
CHR$(9)	Forward one
CHR$(10)	Down one
CHR$(11)	Up one
CHR$(13)	To start of line
CHR$(127)	Back one and delete (like <DELETE>)

CHR$ is not only used for moving the cursor around. In fact any printing of text on the screen can be carried out using CHR$. Try this program

```
   10 CLS
   20 PRINT CHR$(72);CHR$(101);CHR$(108);C
HR$(108);CHR$(111);CHR$(32);CHR$(116);CHR$
(104);CHR$(101);CHR$(114);CHR$(101)
   30 END
```

The numbers inside the CHR$ brackets are all codes for the letters that make up the phrase. Every character that your BBC micro can produce has a code number. This is called the character's ASCII code (ASCII is the American standards institute that thought up the codes a long time ago). Text in your programs (but not the keywords) is stored in the BBC in ASCII code. That is how a microprocessor that can only deal with numbers can be made to cope with letters.

RUN this next short program and you'll be treated to a display of all of the characters available on the BBC micro (well, all of them available in MODE 7 anyway) along with their ASCII codes.

```
10 CLS
20 FOR N=32 TO 126
30   PRINT; N,CHR$(N)
40   NEXT N
50 END
```

How about that? The first program with no new keywords! All the visible characters lie between ASCII codes 32 to 126. Most of the rest are various control codes like those we have already met. Add these lines to your program and it will tell you the code of any character that you type in.

```
10 CLS
20 FOR N=32 TO 126
30   PRINT; N,CHR$(N)
40   NEXT N
41 REPEAT
42   INPUT "CHARACTER ",C$
43   PRINT; ASC(C$),C$
44   UNTIL C$=""
50 END
```

It's a REPEAT-UNTIL loop this time. Line 42 INPUTs a string called C$ from you. Rather than just present you with a question mark (not very friendly) you can write an INPUT statement to present the user with a proper written prompt. This is simply enclosed in inverted commas after the INPUT as it is here.

Line 43 prints out the character that you have typed in and its ASCII code. The code is obtained from the ASC command. This is a simple Basic function that returns the ASCII code of the string inside the brackets. If the string is more than a single character, the code of the first character is returned. ASC is really the reverse of CHR$.

The loop will continue until you press the <RETURN> key. It is stopped by line 44.

There are actually 256 different values that the ASCII codes can take. Only 102 are taken up by the letters, symbols and cursor movement. So what do the ones that we have not yet looked at actually do? The answer you will be pleased to hear, is that they control the colour and graphics in mode 7.

Remember that you typed in MODE 7 at the beginning of this chapter. That is the display mode that your BBC is using at this minute. We will continue to use that mode for the rest of this chapter. In the next chapter we will start to look at the real graphics that are available in the other modes. Meanwhile let's be satisfied with mode 7.

Mode 7 uses a peculiar system to store the data required for the screen in the micro's memory. It is what is known as a 'serial attribute' system. That is, all details such as colours are stored as

special control characters on the screen just like the characters themselves. The control codes are not seen on the screen but they do take up room. RUN this program and you'll see what this means

```
   10 CLS
   20 PRINT CHR$(129);"RED"
   30 PRINT CHR$(130);"GREEN"
   40 PRINT CHR$(131);"YELLOW"
   50 PRINT CHR$(132);"BLUE";CHR$(133);"MA
GENTA";CHR$(134);"CYAN"
   60 END
```

The numbers after the CHR$s control the codes that turn all further text to one colour or another. Line 50 shows how redefining the present text colour cancels the previous colour. Notice how the control code for magenta (133) actually takes up room between the words 'Blue' and 'Magenta'. The code is invisible so this space appears as just that, an empty space. Similarly the 134 code creates a space between 'Magenta' and 'Cyan'.

Along with black and white these are all the colours that the BBC micro can produce. Magenta is the mauve and Cyan the light blue colour.

The text background colour can also be set. For this, however, two codes are required. First a colour must be chosen with one of the codes that we have already met and then a special code (157) is given which switches the background to that colour. Like this

```
   10 CLS
   20 PRINT CHR$(129);CHR$(157);CHR$(131);
"YELLOW ON RED"
   30 END
```

Code 129 selects red, 157 selects a new background of this colour, then 131 selects a new colour (yellow) for the foreground. Notice that the codes for all this take up no less than three character positions.

All the codes affect any text that follow them until the end of the line is reached. A new line is always started in white text on a black background. All the codes, both control codes and ASCII codes, can be found in the tables on pages 486–489 of the User Guide.

There are other codes as well. 136 flashes whatever follows it. Code 137 cancels this to restore a steady display to the rest of that line. Code 152 conceals whatever follows it.

Codes 141 and 140 switch between double and normal height. As the double height characters take up two lines of the display, anything to be printed out in double height must be printed on two successive lines, complete with control codes before both lines. All

the various text control codes are demonstrated in the following program.

```
   10 CLS
   20 PRINT TAB(0,5);CHR$(129);"RED TEXT O
N :"
   30 PRINT CHR$(130);CHR$(157);CHR$(129);
"GREEN"
   40 PRINT CHR$(131);CHR$(157);CHR$(129);
"YELLOW"
   50 PRINT CHR$(132);CHR$(157);CHR$(129);
"BLUE"
   60 PRINT CHR$(152);"CONCEALED";
   70 PRINT CHR$(135);"NOT NOW!"
   80 PRINT CHR$(134);CHR$(136);"FLASHING"
   90 PRINT CHR$(133);CHR$(141);"DOUBLE HE
IGHT";
  100 PRINT CHR$(133);CHR$(140);"BUT NOT F
OR LONG !"
  110 PRINT CHR$(129);CHR$(141);"DOUBLE HE
IGHT"
  120 END
```

Notice how all the lines of the display without any control codes at their start remain in the default colours of white on black.

Although these control codes are a little bit complicated to use, and they riddle the display with holes, the screen as a whole uses very little memory. This means that there is plenty left for your program. This is why very long programs like adventure games, that don't require complex graphics, use mode 7.

Mode 7 is in fact exactly the same as teletext displays on television. If you've seen these you'll know that these displays often include pictures. Indeed graphics is also available in mode 7, and of course it requires control codes! Try this

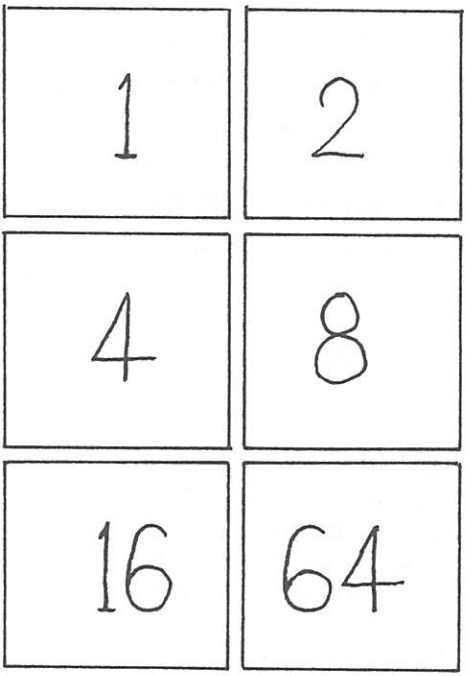
```
   10 CLS
   20 PRINT CHR$(147) "write anything you
care to in small letters only"
   30 END
```

Not what you expected! The code 147 selects the colour yellow just like code 131. However 147 selects yellow graphics and not yellow text. All of the ASCII codes for the alphabet serve two purposes. After any text control code, letters are displayed. After a graphics control code however these graphics shapes are printed onto the screen.

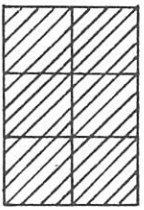
Only the small letters are replaced with graphics characters. The capitals remain ready for use without 'switching back' to text.

Choosing which code to use to put any given shape on the screen is not easy. It involves a bit of maths.

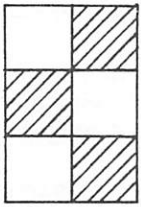
The graphics characters are all made up of six square blobs. These blobs are either 'on' or 'off'. An 'on' blob will appear in the foreground colour and an 'off' blob in the background colour. A solid block the size of a character has all six blobs on. A space has them all off. To find the code required for any combination of blobs off and blobs on, we must imagine that there is a number associated with each blob, like this



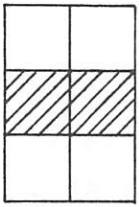
To find the code simply add together the numbers for the 'on' blobs and then add 160. Lets have a look at some examples:



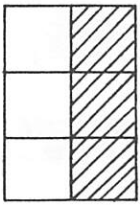
CODE=(1+2+4+8+16+64)+160=255



CODE = (2 + 4 + 64) + 160 = 230



CODE = (4+8) + 160 = 172



CODE = (2 + 8 + 64) + 160 = 234

('ON' Blobs are here shown shaded)

FIG 4.3

Using combinations of these graphics characters, quite complex shapes can be built up, in glorious technicolour, and using very little memory.

Graphics background colour is set in the same way as that for text, using the same code (157). Similarly graphics can be made to flash using code 136 and vanish using code 152.

The use of the graphics characters is exactly the same as the text, only one of the graphics colour select codes is used rather than a text colour select code.

Double height graphics is not possible, but the graphics has a couple of its own special codes all to itself.

Normally graphics in mode 7 is in a 'contiguous' form. That is each blob inside a character and each character joins up with the next. Using the code 154 separates each blob with a small strip of background colour. Code 153 puts it back again.

Code 158 is also only for use with graphics. This is known as the 'graphics hold' code. This fills in the gaps in between graphics characters on the screen, caused by other codes lurking invisibly there, with whatever graphics character is to the left of the gap. This enables you, say, to change the colour of a picture half way across it without riddling it with gaps.

The next program demonstrates all these graphics codes.

```
   10 CLS
   20 PRINT ''''
   30 PRINT CHR$(151);CHR$(157);CHR$(145);
STRING$(30,CHR$(230))
   40 PRINT CHR$(147);CHR$(157);CHR$(145);
STRING$(30,CHR$(185))
   50 PRINT CHR$(146);CHR$(157);CHR$(145);
STRING$(30,CHR$(230))
   60 PRINT CHR$(148);CHR$(157);CHR$(145);
STRING$(30,CHR$(185))
   70 PRINT ''
   80 PRINT CHR$(149);CHR$(243);"CAPITALS"
;CHR$(243);"AMONGST";CHR$(243);"GRAPHICS";
CHR$(243)
   90 PRINT
  100 PRINT CHR$(149);CHR$(243);"doesn't w
ork";CHR$(243);"with";CHR$(243);"small let
ters";CHR$(243)
  110 PRINT
  120 PRINT CHR$(147);CHR$(154);" SEPARATE
D ";STRING$(5,CHR$(255));
  130 PRINT CHR$(147);CHR$(153);"CONTIGUOU
S ";STRING$(5,CHR$(255))
  140 PRINT
  150 PRINT CHR$(150);"NORMAL COLOUR CHANG
E ";STRING$(6,CHR$(225));CHR$(145);STRING$
(10,CHR$(225))
  160 PRINT
  170 PRINT CHR$(150);"WITH GRAPHICS HOLD"
;CHR$(158);STRING$(8,CHR$(225));CHR$(145);
STRING$(10,CHR$(225))
```

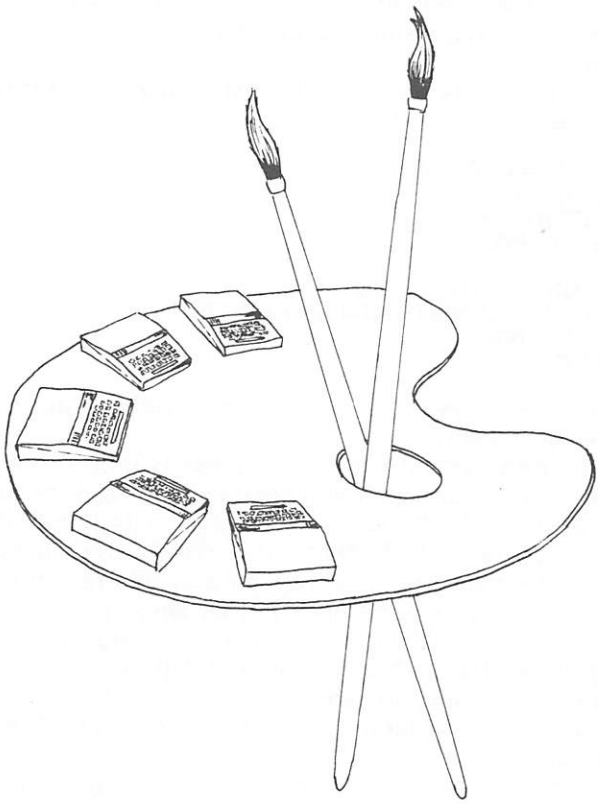
All this has been pretty hard work. So we'll leave this chapter with a bit of (relevant) light relief. Type in this program and RUN it. It's not quite Walt Disney, but it is amusing. Study it well, and you'll be in good stead for the next chapter.

```
   10 MODE 7
   20 PRINT TAB(15,5);CHR$(149);CHR$(248);
CHR$(255);CHR$(253);CHR$(176)
   30 PRINT TAB(15);CHR$(145);CHR$(250);CH
R$(251);CHR$(251);CHR$(176)
   40 PRINT TAB(15);CHR$(145);CHR$(235)
   50 PRINT TAB(14);CHR$(150);CHR$(248);CH
R$(252);CHR$(191);CHR$(253);CHR$(252);CHR$
(176)
   60 PRINT TAB(12);CHR$(150);CHR$(224);CH
R$(254);CHR$(191);CHR$(255);CHR$(247);CHR$
(255);CHR$(191);CHR$(255)
   70 PRINT TAB(12);CHR$(150);CHR$(234);CH
R$(183);SPC(1);CHR$(255);CHR$(253);CHR$(25
5);CHR$(181);CHR$(162);CHR$(175);CHR$(161)

   80 PRINT TAB(12);CHR$(150);CHR$(234);CH
R$(181);SPC(1);CHR$(234);CHR$(255);CHR$(25
5)
   90 PRINT TAB(12);CHR$(145);CHR$(234);CH
R$(165);CHR$(148);CHR$(250);CHR$(255);CHR$
(255);CHR$(176)
  100 PRINT TAB(12);CHR$(147);CHR$(234);CH
R$(148);CHR$(232);CHR$(255);CHR$(167);CHR$
(239);CHR$(253)
  110 FOR N%=1 TO 3
  120    PRINT TAB(12);CHR$(147);CHR$(234);
CHR$(148);CHR$(234);CHR$(255);SPC(1);CHR$(
234);CHR$(255)
  130    NEXT N%
  140 PRINT TAB(12);CHR$(147);CHR$(234);CH
R$(232);CHR$(252);CHR$(255);SPC(1);CHR$(23
4);CHR$(253);CHR$(252)
  150 FOR Y%=18 TO 24
  160    PRINT TAB(0,Y%);CHR$(146);CHR$(157
);
  170    NEXT Y%
  180 REPEAT
  190    PRINT TAB(17,7);CHR$(242);CHR$(250
);CHR$(161);SPC(3);CHR$(232);CHR$(255)
  200    PRINT TAB(21,8);SPC(1);CHR$(224);C
HR$(254);CHR$(183)
  210    PRINT TAB(21,9);CHR$(252);CHR$(255
);CHR$(167)
  220    T%=TIME+30:REPEAT UNTIL TIME>T%
  230    PRINT TAB(17,7);CHR$(243);CHR$(251
);CHR$(161);SPC(1);CHR$(250);CHR$(181);SPC
(2)
  240    PRINT TAB(21,8);CHR$(234);CHR$(181
);SPC(2)
  250    PRINT TAB(21,9);CHR$(254);CHR$(181
);SPC(1)
  260    T%=TIME+30:REPEAT UNTIL TIME>T%
  270    UNTILFALSE
  280 END
```

Chapter Five

Graphics



So far, in this book we have almost exclusively used the mode 7 display. Now we are going to explore the other display modes available with the BBC micro.

There are eight display modes available, numbered zero to seven. Mode 7 is unusual in that it is compatible with teletext transmission displays. The others are all different to this but similar to one another. The modes differ in the number of characters that can be displayed on the screen at once, the number of colours that can be displayed, and whether graphics can be displayed or not.

The reason for having several modes available is one of memory. It would be possible to have a single display with lots of characters on the screen, lots of colours, and graphics — the best of all worlds. Unfortunately your BBC must store the data from which the display is made in its memory. The more complex the display, the more memory is taken up, and the less there is left for your actual program. The compromise adopted on the BBC micro is to have several formats of display, each with its own advantages and disadvantages, so that a suitable one can be chosen for a particular application.

The following short program demonstrates the text format of the various modes. Try this out.

```
10 FOR M%=0 TO 6
20   MODE M%
30   GOSUB 60
40   NEXT M%
50 END
60 FOR N%=1 TO 1000
70   PRINT CHR$(RND(60)+32);
80   NEXT N%
90 RETURN
```

Some of these keywords you have met before, but some you haven't. The worst is GOSUB.

One of the most useful features of Basic is that of subroutines. Frequently in a program we want to carry out some particular action a number of times. If this action is to be repeated time after time, then using loops, either FOR–NEXT or REPEAT–UNTIL loops, is fine. However, if the action is to be repeated at different times in the program, with other actions occuring between these times, then loops are no use to us. Of course we could just write out the relevant bit of program each time that it is needed, but this can be very time and memory consuming. Instead we need a method of calling upon a particular bit of a program time and time again. That is what subroutines are all about.

A subroutine is a program all in itself, that is called upon by the main program to do its bit. In this example we are dealing with here,

the subroutine is in lines 60 to 90. Notice how this part of our program is put after the END statement.

The subroutine is brought into action, or 'called', by the command GOSUB in line 30. The number following the GOSUB is the line number where the subroutine starts. In this case it is 50. Incidentally you'll be pleased to know that the GOSUB numbers are all taken care of when you use the RENUMBER command. You don't have to go through a renumbered program to change all these.

At the end of the subroutine there is the command, RETURN. This is nothing to do with the key, <RETURN>. This command tells the micro that the subroutine is finished and it is to return to the main program where it left off. The micro will then continue through the main program quite normally.

In this example the subroutine is one to print out onto the screen a lot of random characters. It does this using a FOR–NEXT loop and the CHR$ that we met in the last chapter. The ASCII codes that it uses are generated by the RND function.

A program can use any number of subroutines. Each subroutine can be called any number of times. In fact subroutines can call other subroutines. This is known as 'nesting', in the same way as we saw loops being nested in the last chapter.

Now that you are thoroughly familiar with subroutines we are going to forget them! We are going to replace them with an entirely different programming form, procedures.

Subroutines date back to the early days of Basic. They are very nice as far as they go, but they are not very versatile and do make reading a program very difficult as you are always hopping around with no idea of where you are going. Procedures are a more modern invention and go a long way to curing all that. There are many instances where subroutines and procedures can be used interchangeably. As BBC Basic is fortunate enough to be blessed with procedures, these will be used exclusively from now on, unless there is a good reason not to.

To start the ball rolling, re-write the last program using proceedures, like this

```
 10 FOR M%=0 TO 6
 20   MODE M%
 30   PROCFILL_SCREEN
 40   NEXT M%
 50 END
 60 DEFPROCFILL_SCREEN
 70 FOR N%=1 TO 1000
 80   PRINT CHR$(RND(60)+32);
 90   NEXT N%
100 ENDPROC
```

The idea is still very similar. The procedure is called from line 30, by name. No line numbers now. Any name, up to ten characters long, can be given to a procedure. Spaces can't be used but the underline character does quite nicely instead, as you can see from our procedure name here, FILL_SCREEN. The procedure itself lies after the END statement just like the subroutine.

The start of the bit of program that actually makes up the procedure is signalled by the keyword, DEFPROC. This is short for 'define procedure', and that is just what you are doing. The end of the procedure is marked with an ENDPROC. In between we have the same program lines as we used for our subroutine.

Anyway that's enough of procedures for the minute. We will be meeting them again many times in the future. Meanwhile let's get back to display modes.

RUN his last program and you'll be presented with a screen full of garbage in every mode. The different display modes are called with the command, MODE, in line 20, followed by the mode number. In this case the number is provided by the changing loop variable, M%. You can see the different numbers of characters that fit onto a screen in the different modes. The size of the characters and their proportions, of course change also. The text format of the eight modes in number of columns by number of rows is given below.

Mode 0	80 x 32
Mode 1	40 x 32
Mode 2	20 x 32
Mode 3	80 x 25
Mode 4	40 x 32
Mode 5	20 x 32
Mode 6	40 x 25
Mode 7	40 x 25

Now you can see that this is a little repetitive. Each format appears twice. In the case of modes 6 and 7 this is because mode 7 is special as we have already seen. It is the teletext mode.

In the case of the other pairs (0 and 3, 1 and 4, 2 and 5) the reasons are more complex. Mode 0 is capable of displaying graphics, whereas mode 3 is not, mode 2 can display 16 colours whereas

mode 5 can only display 4 at a time. mode 4 can similarly only display two colours at a time, but mode 1 can display 4 colours. The full description of the different modes, then, would be this

Mode	Text	Graphics	Colours
0	80 x 32	640 x 256	2
1	40 x 32	320 x 256	4
2	20 x 32	160 x 256	16
3	80 x 25	—	2
4	40 x 32	320 x 256	2
5	20 x 32	160 x 256	4
6	40 x 25	—	2
7	40 x 25	teletext	8

The use of all these modes for text is almost exactly the same as we have already seen for the case of mode 7. That is, a semicolon in a PRINT statement will cause the printed characters to follow on directly from the last, TAB is used to place characters anywhere on the screen, ASCII codes are used with CHR$ to produce characters from figures and other cursor control codes.

There are some differences however. For a start, a greater number of 'fields' (remember them?) can be fitted across the screen in some modes. The limits for the co-ordinates in the TAB command are higher in some modes because there can be more characters on the screen.

However the biggest difference is the use of colour and graphics. In all the modes, 0 to 6, the control codes that we used in mode 7 to change the colour of the following text on the line, and to make it flash, and double height, and graphics, and separated, and contiguous, and concealed, and so on, no longer apply. The cursor control codes still work, but colour and graphics are controlled in an entirely different way. That's what we're going to look at now.

In these modes there need be no holes riddling your text and graphics. This is because the system used is a 'parallel attribute' system. That is: the data telling the video circuitry in your computer what colours to make the text is stored in parallel with the data telling it what shape the text (or graphics) is. In mode 7 this information was stored in series with the shape data as the other, invisible, characters. The trouble with the parallel system is that it takes up a good deal more memory.

All we have to do to put colour on the screen in modes 0 to 6 then is to tell the micro what we want. Switch to MODE 2 (as this mode can support all 16 colours) by typing just that and then type in, in succession

```
COLOUR 1
COLOUR 2
COLOUR 3
COLOUR 4
COLOUR 5
COLOUR 6
COLOUR 7
COLOUR 8
COLOUR 9
COLOUR 10
COLOUR 11
COLOUR 12
COLOUR 13
COLOUR 14
COLOUR 15
```

Coloured text as easy as that. Mode 2 has 16 colours available on the screen at once. In fact there are only eight colours available on the BBC micro. Eight of the mode 2 colours are flashing versions of other colours. Don't forget that black is counted as a colour. Type COLOUR 0 and you'll see this. Anything further that you type is on the screen, but you can't see it because it is printed in black on a black background.

The background colour can be changed just as easily. Try this

```
COLOUR 132
COLOUR 137
```

You can find the number required for a background colour by adding 128 to the number required to get that colour as the foreground colour. In other words COLOUR 1 gives a red foreground and COLOUR 129 gives a red background.

In the other modes these numbers do not relate to the same colours. For a start there are not the same number of colours available in any of the other modes. In all of them, however, number 0 is black and the highest number white. The default colours in all

modes are black for the background and white for the foreground.

In the other modes the colour codes give colours according to this scheme:

Colour codes	Modes 0,3,4,6	Modes 1,5	Modes 2
0 128	Black	Black	Black
1 129	White	Red	Red
2 130		Yellow	Green
3 131		White	Yellow
4 132			Blue
5 133			Magenta
6 134			Cyan
7 135			White
8 136			Black/White
9 137			Red/Green
10 138			Green/Magenta
11 139			Yellow/Blue
12 140			Blue/Yellow
13 141			Magenta/Green
14 142			Cyan/Red
15 143			White/Black

We can demonstrate all of these colours in all of these modes with this program. Before you type it in, make sure that you are using a colour scheme that you can read. Set the colours back to white on black.

```
 10 FOR M%=0 TO 6
 20   MODE M%
 30   PRINT TAB(10,10)"MODE ";M%
 40   FOR C%=0 TO 15
 50     T%=TIME+150:REPEAT UNTIL TIME>T%
 60     COLOUR 128+C%
 70     CLS
 80     NEXT C%
 90   PRINT CHR$(7)
100   NEXT M%
110 END
```

More FOR–NEXT loops. This time we have two loops 'nested' together. Just as subroutines and procedures can be nested, so can loops. Each time the program goes around the 'outer' loop, it does a whole set of inner loop operations.

In this program the inner loop cycles through all the colours. The whole screen is filled with one colour at a time. This is done by setting the background colour to a value (line 60), and then clearing the screen with a CLS command (line 70). The COLOUR command effects every printing command after it. As clearing the screen is effectively printing the background over the whole screen, so it is a quick way of turning the whole screen to a certain colour.

Each time around the inner loop, of course, the value used is changed. A time delay, as we've used before, is included in line 50 so we can see what is going on. Try the program without this line if you want to see the speed of this operation!

The outer loop, of course, just switches between the modes in turn. It changes the mode after every 16 colour changes. Just after each mode is through a beep is produced to tell you that the next is starting. This is produced by the ASCII code 7 'printed' in line 90. This is one of the control codes that is common to all of the modes, 0 to 7. At the start of each time around the outer loop the mode name is printed on the screen (line 30).

In mode 2 this is all very well. There are 16 colours and we are going around the inner loop 16 times. But what about the other modes with only two and four colours? How do they react to us demanding colours that don't exist?

Fortunately (for this and a lot of other programs) in each mode the colour codes can be thought of as repeating themselves again and again for ever. That is to say in mode 3 (a two colour mode) code 1

is not the only number to give the colour white, but 3, 5, 7, 9, 11, 13, 15 and so on, also give this colour. In mode 5 (four colour) red is given if you use COLOUR 1 or 5 or 9 or 13.

This means that in our program we can happily call 16 different colours in the inner loop in all the modes without harm. RUN the program and you'll see all the colours produced.

That's enough about text for the minute. If you're anything like most BBC micro owners, you'll be itching to get your hands on some graphics.

Try this program. It's tedious to type in, but unfortunately that's what graphics on the BBC is often all about!

```
  10 MODE 2
  20 COLOUR 130
  30 CLS
  40 GCOL 0,0
  50 MOVE 300,200:DRAW 900,200
  60 DRAW 900,600:DRAW 300,600:DRAW 300,2
00
  70 GCOL 0,1
  80 MOVE 300,600
  90 DRAW 600,800:DRAW 900,600
 100 GCOL 0,7
 110 FOR X%=350 TO 770 STEP 140
 120   MOVE X%,450:DRAW X%+70,450:DRAW X%
+70,530
 130   DRAW X%,530:DRAW X%,450
 140   NEXT X%
 150 FOR X%=350 TO 770 STEP 420
 160   MOVE X%,300:DRAW X%+70,300:DRAW X%
+70,370
 170   DRAW X%,370:DRAW X%,300
 180   NEXT X%
 190 GCOL 0,6
 200 MOVE 550,200:DRAW 550,370
 210 DRAW 650,370:DRAW 650,200
 220 GCOL 0,3
 230 MOVE 550,200:DRAW 0,0
 240 MOVE 650,200:DRAW 1279,0
 250 END
```

Line 10 selects display mode 2. This is the graphics mode with all sixteen colours available. Lines 20 and 30 fill the screen with a green background in the same way as the last program.

The form of commands (or 'syntax') is quite simple. MOVE will take the cursor to a certain point in the same way as TAB. The difference is that it is a different cursor!

Up until now we have dealt exclusively with the Text cursor. We are now dealing with the Graphics cursor. Text is printed at the text

cursor and graphics is drawn at the graphics cursor. The text cursor can be seen as a flashing line at the cursor position. The graphics cursor cannot be seen, but it can be moved around.

The co-ordinate system used for graphics is also different to that used for text. The text co-ordinate system varies depending on the mode. Different modes have different numbers of characters across and down the screen so the limits of the co-ordinates differ from mode to mode. When using graphics, however, although the graphics resolution differs from mode to mode (have another look at that table further back in this chapter), the co-ordinate system stays the same. The graphics co-ordinate system always goes between 0 and 1279 horizontally and 0 and 1023 vertically.

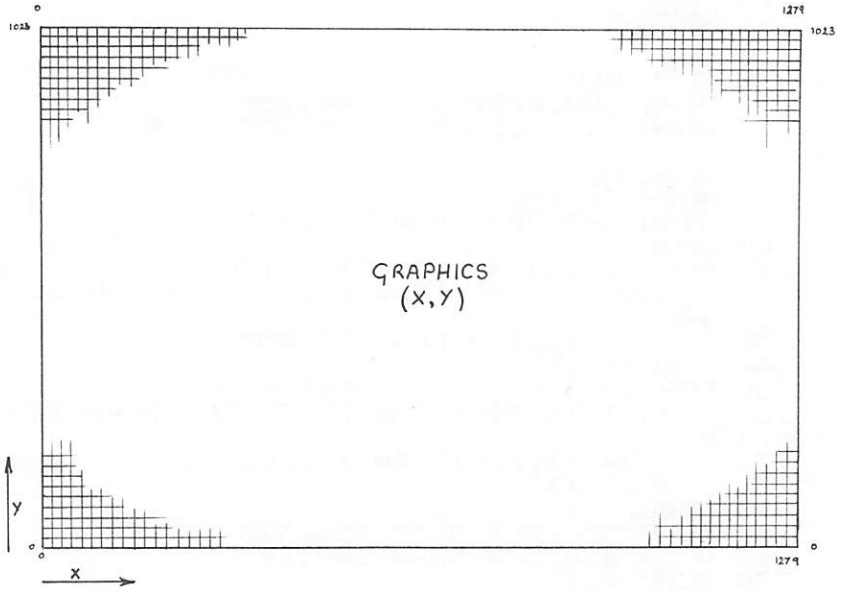


Fig. 5.4

Another difference to the text co-ordinate system that is worth noting is that the origin, point (0,0), is at the bottom left corner for graphics and not at the top left as it is for text.

MOVE, then, moves the graphics cursor to the (x,y) co-ordinates that follow the command. When you change mode the cursor is always put to the origin in the bottom left hand corner, just as the text cursor is moved to the top left.

DRAW does just that. The cursor is moved to the new point and a line is drawn between the two in the current graphics foreground

colour. The current graphics colour is changed with the command GCOL (for Graphics COLour).

There are two numbers following the GCOL command. For the moment we will only look at the second. This denotes the colour. The number used here is the same as that required for a text colour change.

So that's it. Most of this last program simply moved the graphics cursor around the screen between some cleverly chosen points, drawing lines as it went. Follow the MOVE and DRAW commands on a piece of squared paper and you'll see how the picture is built up.

Of course, as this is the BBC micro, it doesn't end there. Lines can be drawn in other ways too. MOVE and DRAW are just convenient forms of the two most commonly used versions of the command PLOT. PLOT has no less than three numbers following it. The last two are the x and y co-ordinates, as in MOVE and DRAW. The first of the numbers controls what form of plotting the micro is to carry out. These are summarised here.

PLOT 0	Move relative
PLOT 1	Draw line relative, in foreground colour
PLOT 2	Draw line relative, in inverse colour
PLOT 3	Draw line relative, in background colour
PLOT 4	Move absolute
PLOT 5	Draw line absolute, in foreground colour
PLOT 6	Draw line absolute, in inverse colour
PLOT 7	Draw line absolute, in background colour

FIG 5.5

'Relative' means that the x and y co-ordinates are taken as values to be added on to the x and y co-ordinates of the graphics cursor position. 'Absolute' means that the co-ordinates are taken to be based on the origin. DRAW is equivalent, then, to PLOT 5 and MOVE to PLOT 4.

The foreground and background colours mentioned above are fairly straightforward. These relate to the colours you have already given in a program using the GCOL command. The 'inverse' colour however needs a bit of explanation.

The really useful feature of graphics on the BBC micro is the way in which lines and shapes can be drawn onto the screen in a manner dependent on what is already up there. A PLOT command drawing

in inverse colour looks at what the line is to pass over. The line is drawn in a colour which is the inverse of the colour there at the moment.

In a two colour graphics mode (0, 4 or 5) this means that over a white background the line comes out black and over black it is white. In the multicolour modes, the resulting colour can be found by looking at the table of colour codes given earlier in the chapter. The inverse of colour 1 is 15, the inverse of colour 5 is 12, the inverse of colour 10 is 7, and so on. This next program demonstrates this inverse drawing.

```
 10 MODE 2
 20 FOR C%=0 TO 18
 30   POSITION%=C%*80
 40   PROCBAND(POSITION%,C%)
 50   NEXT C%
 60 PLOT 4,0,500
 70 PLOT 6,1279,500
 80 END
 90 DEFPROCBAND(X%,C%)
100 GCOL 0,C%
110 MOVE X%,0
120 MOVE X%+80,0
130 PLOT 85,X%,1024
140 PLOT 85,X%+80,1024
150 ENDPROC
```

RUN the program and you'll see a single line drawn over several bands of colour. The line is drawn in the inverse of each band colour as it passes over it.

Line 60 moves the graphics cursor to a point half way up the screen at the left hand edge using the PLOT 4 command. This could be replaced with a MOVE command (try it!). Line 70 plots an inverse line across to a point on the opposite side of the screen. The rest of the program draws the coloured bands. Let's have a look at how this is done.

A procedure is used. This is called PROCBAND. In this program we have expanded our idea of procedures a little. You will notice that after the call to the procedure (PROC) there is a set of brackets containing the names of two variables (POSITION% and C%). At the start of the procedure definition there are similar brackets containing X% and C%. What is happening is this: Each time that the procedure is called, the value assigned to the variable POSITION%

at the time is now assigned to X% while the procedure is in operation. The same thing is happening with C%.

Using this system, numbers can be easily passed to procedures from the main part of the program.

Inside the procedure we also have a new idea. We are using PLOT 85. As well as the eight numbers to follow PLOT that we have seen so far, there are in fact 56 others! Fortunately these can be summed up quite easily.

Numbers 8 to 15 correspond to 0–7 that we have already looked at insofar as relative co-ordinates, colours used, and so on goes. If these numbers are used however, the line is drawn without the very last point. This is useful when you want a line of one colour to meet another of another colour without 'overdrawing' it.

Numbers 16–23 also correspond to 0–7 but these produce a dotted line. 24–31 produce a dotted line without the last point.

Numbers 64–71 produce only the last point of a line. PLOT 69, then, produces a single point in the foreground colour at co-ordinates (x,y).

Numbers 80–87 also correspond to 0–7, but these produce a solid triangle between the new point and the last two points visited. It is this feature that we were using to produce our coloured bands.

Lines 110 and 120 in our program, then move the graphics cursor to two points in succession without drawing any lines. These points are along the base of the screen at X% (which has been passed to the procedure from the FOR–NEXT loop) and at X%+80. The figure 80 was chosen as the width of the bands so that we could fit 16 of them across the full screen (1280).

In line 130 a PLOT 85 command is used. This fills in a triangle between a new point that is given in absolute co-ordinates, and the last two points visited, in the current graphics foreground colour. Now the colour has been set in line 100 as the colour corresponding to C%. C% is of course changing each time we call the procedure.

The new point in this PLOT 85 command is one at the top of the screen, directly above the point we visited in line 110. So a triangle is created between the three points (visited in lines 110, 120 and 130) in colour, C%.

When we come to the next PLOT 85 command the new point is one directly above the point we MOVED to in line 120. A second triangle is drawn. This time however the two last visited points are now the points visited in lines 120 and 130 so the triangle is positioned to finish off the rectangle formed by all four points.

If this seems confusing to you, then read through the last few paragraphs again. Remember that the triangle is plotted between the new point and the LAST TWO points. Put a short delay into the program like this

```
135 T%=TIME+50:REPEAT UNTIL TIME>T%
145 T%=TIME+50:REPEAT UNTIL TIME>T%
```

Now you can see each triangle as it is drawn.

You may think that drawing triangles in this way has not got a great deal of application. However remember that any shape you may care to dream up can be made up of triangles of different sizes and in different orientations.

There is another method of drawing solid shapes. This relies on a method very similar to that which we humans use when drawing or painting. When you are painting a solid area of colour on a piece of paper, you would probably draw the outline first and then fill in the area that this encloses.

Drawing outlines with the BBC micro we have already covered. Now lets have a look at how to fill them in.

There are still some PLOT types that we haven't covered: these are the fill commands. Numbers 72–79 correspond to the 0–7 that we have looked at. With these PLOT commands however the graphics cursor is moved to the point (x,y) and then a search is made to the left of this point until the first non-background colour point is reached. A similar search is made to the right and then a line is drawn between the two non-background points.

By repeating this operation at a series of vertical positions inside an outline it is possible to fill in the outline with colour.

PLOT 88 to 95 do a similar task but this time only the right hand side is searched. A line is then drawn between that point and (x,y) in the background colour. This can be used for 'undrawing' objects.

Filling shapes in this way is a complicated operation to use. You won't find any explanation of these PLOT commands in the User Guide. However it remains the best and quickest method of filling complex shapes. The final program demonstrates its use on a weird and wonderful shape that would be very difficult to fill using triangles, unless you wanted to do an awful lot of plotting out shapes on graph paper!

```
 10 MODE 1
 20 GCOL 0,1
 30 MOVE 600,100
 40 DRAW 300,250
 50 DRAW 580,450
 60 DRAW 60,500
 70 DRAW 300,650
 80 DRAW 500,800
 90 DRAW 550,900
100 DRAW 800,900
110 DRAW 700,700
120 DRAW 1000,600
```

```
130 DRAW 700,400
140 DRAW 900,100
150 DRAW 600,100
160 T%=TIME+100:REPEAT UNTIL TIME>T%
170 GCOL 0,2
180 FOR Y%=100 TO 900 STEP 4
190   PLOT 77,600,Y%
200   NEXT Y%
210 END
```

Most of this should be self explanatory. The first part of the program draws the complicated outline. The FOR–NEXT loop repeats a filling PLOT command down the whole length of the shape to fill it all in. Pretty neat, huh.

The filling PLOT command is repeated at intervals of 4 up the height of the shape as this is the height of a single drawn line. The height of the smallest possible part of the screen. Using a bigger figure will give you a striped shading and a smaller step will just slow down the process as each fill line is drawn more than once. Try 12 and 1 as the step. Put in a delay, like this, to show the filling process more clearly.

```
195 T%=TIME+30:REPEAT UNTIL TIME>T%
```

This chapter has been pretty hard work. We have covered all the simple aspects of using the modes 0 to 6. There is still a long way to go yet, though. We have still to look at the delights of animation and other advanced uses of the display.

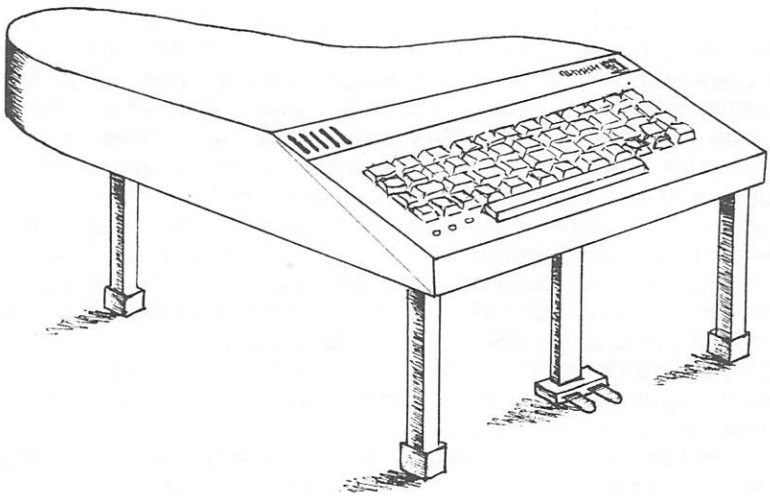
In the next chapter, however we are going to take a break from the TV screen for a while and concentrate on an area that we have largely ignored so far — sound. Before we go, however, here is a little program to finish off with. Most of the concepts that we have met are contained in this program, and it will help you to understand them by way of repetition. It may also even give you a laugh.

```
 10 MODE 1
 20 PROCBACKGROUND
 30 PROCGLASS
 40 PROCFILL
 50 PROCBUBBLES
 60 END
 70 DEFPROCBACKGROUND
 80 GCOL 0,1
 90 MOVE 0,0
100 MOVE 1279,0
110 PLOT 85,0,300
120 PLOT 85,1279,300
130 ENDPROC
140 DEFPROCGLASS
150 GCOL 0,3
```

```
160 MOVE 550,300:DRAW 750,300
170 DRAW 670,320:DRAW 630,320
180 DRAW 550,300
190 MOVE 630,320:DRAW 630,500
200 PLOT 1,-20,20
210 PLOT 1,-140,40
220 PLOT 1,-20,20
230 PLOT 1,400,0
240 PLOT 1,-20,-20
250 PLOT 1,-140,-40
260 PLOT 1,-20,-20
270 DRAW 670,320
280 ENDPROC
290 DEFPROCFILL
300 FOR Y%=300 TO 570 STEP 4
310   PLOT 77,650,Y%
320   T%=TIME+6:REPEAT UNTIL TIME>T%
330   NEXT Y%
340 DEFPROCBUBBLES
350 FOR N%=1 TO 25
360   X%=RND(400)+450:Y%=RND(200)+580
370   PLOT 69,X%,Y%
380   NEXT N%
390 ENDPROC
```

Chapter Six

Sound advice



The BBC micro is blessed with just about the most sophisticated sound producing system of any personal computer on the market. Unfortunately it is not one of the easiest to use. This is partially due to the very complexity of the system.

There are only two main Basic keywords used to control the sound production on your BBC micro. These are SOUND and ENVELOPE. SOUND is the one that actually tells the micro to produce a noise and is also the simplest to understand. We'll deal with this one first.

There are four sound producing 'channels' in your micro. Three of these can produce notes and the other noises (often called 'white' noise). Producing noises is a little different so we'll concentrate on notes first.

To produce a note with the SOUND command we must tell the computer several things: the number of the channel from which we want to sound to come, its volume, its pitch and the length of time the note is to last.

This information is given to the computer by way of numbers following the SOUND statement. The numbers must be in the correct order and they must be separated by commas, like this

```
SOUND C,V,F,D
```

For notes the channel number must be 1, 2 or 3. Up to three notes can be produced at once on the BBC micro giving simple chords.

The volume number is rather strangely given as a negative number. –15 is the loudest and zero is silent. There is a good reason for this but we'll leave that until we get on to ENVELOPE.

The pitch number must lie between zero and 255. Zero gives the lowest note that your micro can produce and 255 the highest. the numbers do not represent the frequency of the notes, but you will find that an increase of four in the pitch number will give an increase in pitch of one semitone. This means that the smallest musical interval that your BBC can produce is one quarter of one semitone. That should be fine enough for most purposes!

Lastly the duration of the note is measured in 20ths of a second. Any value for the duration can be given between 1 and 254. This gives durations of between 0.05 and 12.7 seconds. Giving the duration a value of 255 will make an endless note that can only be stopped by pressing <ESCAPE> or <BREAK>.

Only whole numbers are allowed as any of the numbers in the SOUND statement. Any real numbers are treated as integers by having their fractional parts stripped off.

The first program of this chapter gives you some idea of the range of notes that can be produced by your BBC micro. It will also show

you how the SOUND statement is used in practice. Use mode 7 for all of the programs in this chapter.

```
10 CLS
20 FOR P%=0 TO 255
30   PRINT P%
40   SOUND 1,-15,P%,6
50   NEXT P%
60 END
```

A simple FOR-NEXT loop that produces a note at every pitch possible on your micro. When you RUN this program, however, you'll find that all the pitches are printed on the screen by line 30 before the sounds are finished. This demonstrates a feature of the BBC sound facilities that is both irritating and tremendously useful.

Once a program is running, and the micro comes across a SOUND statement it simply passes all the relevant numbers over to the sound producing circuitry to join a queue of sounds to be produced. This means that a command for a long sound will not hold up the program. In our program here this means that all these sounds that take a long time to produce will not hold up the rest of the program — the print statements.

In this program we have only used a single channel. Let's change it to produce simple chords.

```
41 SOUND 2,-15,P%+40,6
42 SOUND 3,-15,P%+80,6
```

The channels two and three are now producing notes exactly five tones (40 pitch units) and ten tones (80 pitch units) above the note being produced by channel one. When you run this program you'll find that the run up the scale is fine for most of the range of notes but sounds very odd at the top end.

This is because channels two and three are being told to produce notes that are outside the 0–255 range. In fact what is happening is that these channels then produce low notes. If the pitch number is higher than 255, then 255 is subtracted from it until it comes within the range.

The pitch wraps round the top of the range to come back on the bottom. Giving a pitch number of 258 has the same result as giving one of 3. The same thing happens at the bottom of the range. A pitch of –5 gives the same note as one of 250. It is rather like the way in which the two colours in mode 0, say, are repeated over and over again in the 16 colour codes.

We can demonstrate this wrap around effect with this simple program

```
10 CLS
20 P%=0
30 REPEAT
40    SOUND1,-15,P%,1
50    P%=P%+1
60    UNTIL FALSE
70 END
```

Although the value of P% is continually increasing, the notes coming out of your micro's speaker just cycle through the same range again and again. You'll have to press <ESCAPE> to stop this one.

This is all okay as far as it goes. However, when we produce two notes 'at once', as we were doing, the notes in fact do not start at the same time. Because a program is carried out sequentially, the instructions to produce a second note arrive at the sound chip a little bit after the instructions to produce the first. The gap is too small to hear directly but 'cleaner' chords can be obtained by synchronising the notes to all start at exactly the same time.

This is done using the first number in the SOUND statement, the channel number. Although this number can be the simple single figure type that we have used so far, if we want to do complicated things like synchronisation then we replace it with a four figure number like this

SOUND &HSFC,V,P,D

H	Continuation	0 or 1
S	Synchronisation	0–3
F	Flush queue	0 or 1
C	Channel number	0–3

Fig. 6.1

The ampersand sign (&) is very important. It tells the computer that the following number is given in what is known as hexadecimal. You needn't worry what this really means, just include it.

By changing the values of each figure in this first number we can control the timing of a sound. Making the first figure (H) one means that the sound is used as a dummy note to allow the previous one to continue. This is only really used when the ENVELOPE command is also being used.

Setting the second figure (S) affects the synchronisation of the

sound with others. The value given to S is the number of other channels that this note is to be synchronised with. To play a true three note chord, then, each SOUND statement should have this figure, S, set to two.

F controls the rest of the sound 'queue'. If F is set to 1 then this SOUND statement will be played immediately. Any other sounds waiting in the queue to be played will be flushed from the queue. It is a way of giving one sound priority.

The last of the four figures in our new expanded version of C is C itself. This is our old friend the channel number again. If the other figures (H, S and F) are zero then the whole number reduces to just this last figure. That is what was happening when we only used the C figure earlier in the chapter.

We can demonstrate the use of the expanded first SOUND number with this program. It inputs a number from the keyboard and then produces a sound, with that number as the pitch number. The sound will continue (D=255) until you input a new pitch. The new note will stop the old by using the flush part of the extended SOUND number. This repeats until you press <ESCAPE>.

```
10 CLS
20 REPEAT
30   INPUT "PITCH ",P%
40   SOUND &0011,-15,P%,255
50   UNTIL FALSE
60 END
```

Enough of these pretences to music, let's have a bit of noise! The sound channel on the BBC micro that we have not yet dealt with is numbered zero. This is the noise channel. This channel cannot produce musical notes, only variants of noise. The type of noise produced is controlled by P, the number that controls the pitch of the other channels. This cannot have as many values (256) as it can take for the other channels but must lie between 0 and 7. All the other numbers in the SOUND command have the same affects. The eight values of P have the following effects for channel zero

P	Effect
0	High pitched noise

1 Medium pitched noise
2 Low pitched noise
3 Noise (pitch following pitch of channel one)
4 High pitched periodic noise
5 Medium pitched periodic noise
6 Low pitched periodic noise
7 Periodic noise (pitch following pitch of channel one)

Fig. 6.2

Numbers 0–2 just about explain themselves. You can try them out for yourself in a minute anyway. The periodic noises of 4–6 are the same as 0–2 but they cause oscillating sounds that have a little bit more note-like quality to them (but not much!). The noises caused by numbers 3 and 7 are dependent for their pitch on whatever pitch is currently being played on channel number one. The note on channel one doesn't have to be played at any volume, just so long as it is played. Try this short program for a demonstration.

```
 10 REM ***DEMO OF NUMBERS 0-7***
 20 FOR P%=0 TO 7
 30   PRINT "P% = ";P%
 40   SOUND 0,-15,P%,20
 50   T%=TIME+100:REPEAT UNTIL TIME>T%
 60   NEXT P%
 70 REM ***DEMO OF NUMBER 7***
 75 PRINT "PART TWO"
 80 SOUND 0,-15,7,254
 90 FOR P%=0 TO 255 STEP 2
100   SOUND 1,0,P%,2
110   NEXT P%
120 END
```

The first part of the program just runs through the eight numbers that you can use as P with channel zero and gives you a one second burst of each. The value of P is also displayed on the screen at the same time. Notice how a delay must be put in at line 50 to stop the numbers on the screen running ahead of the sounds as they were in the first program in this chapter. The exact noise that you get for

numbers 3 and 7 depends on what you did to channel one last.

The second part of the program is devoted to noise number seven. Number three has a similar effect but a periodic noise is given. You can try substituting number three for yourself.

The pitch of channel one is increased across its whole range by the FOR-NEXT loop between lines 90 and 110. The volume of the sound produced by channel one (line 90) is set to zero so that you don't actually hear that channel one note. While all this is going on a long noise has been initiated on channel zero by line 80. This is the noise that you hear but its pitch is controlled by the pitch of the silent channel one, and it rises steadily.

A new keyword appears in this program. In fact it appears twice. The word REM stands for 'REMark'. Anything that follows a REM statement is ignored by the computer. This means that you can leave remarks dotted about your program to remind yourself (or anyone else reading the program) what is going on. This is (another) good habit to get into. So far in this book we haven't been using remarks because all of each program has been spelt out in detail in the text anyway. From now on it's going to be different. You're going to have to rely on the REM statements for information on bits of a program that we have already covered.

Although we can produce a wide range of interesting notes and noises with the SOUND command, we still have not produced anything that could be described as 'natural'. Now producing sounds with your BBC that are going to fool anyone is not really possible but we can get a lot nearer than we have done so far.

The problem lies in the way a note changes with time. If you listen to any 'natural' sound (anything from a gently tinkled piano to a car backfiring) you'll notice how the sound varies as it is produced. The volume of the note from a piano rises rapidly in the first fraction of a second and then dies away gradually.

To simulate these kind of effects we can write a complicated program with a whole stream of SOUND commands producing the same note at different volumes to follow the natural pattern. However this is not very sensible. Instead we use a keyword that is designed to do all this for us. This keyword is ENVELOPE. The way in which the properties of a sound vary as it sounds is known as the sound's 'envelope'.

The ENVELOPE command has no less than fourteen (yes, 14!) numbers following it, so it is not the easiest feature of BBC Basic. It is worth mastering, however, because only then will you be able to truly appreciate the best sound facilities of any home computer.

You can define up to sixteen different envelopes at any one time. Each can be used to affect any sound on any channel. The envelopes are numbered 1 to 16. The number of the relevant envelope take the

place of the volume control number, V, in the SOUND statement.

But how to define the envelope in the first place? An envelope is defined using the ENVELOPE command at any point in the program before the SOUND command that is to use it. The ENVELOPE command takes this form:

```
ENVELOPE N,T,PC1,PC2,PC3,PN1,PN2,PN3,A,D,S
,R,AT,DT
```

There, all 14 numbers. About half of these numbers are concerned with changes to the pitch of a note as it is sounded and the other half with changes of volume. To start with we will look at the pitch changes. Let's first then look at the numbers concerned with pitch changes.

N	1 to 16	Envelope number
T	0 to 127	Length of each step in 1/100th sec
PC1	−128 to 127	Change per step in pitch section one
PC2	−128 to 127	Change per step in pitch section two
PC3	−128 to 127	Change per step in pitch section three
PN1	0 to 255	Number of steps in section one
PN2	0 to 255	Number of steps in section two
PN3	0 to 255	Number of steps in section three

Fig. 6.3

Lets take them in order. First N. This is simply a label for your envelope. You can use 16 envelopes at once as long as you are not using another Basic keyword, BPUT#. Then you can only use four.

The note is split into tiny steps. The pitch of the note is changed (as determined by the ENVELOPE) between each step. The length of each step in hundredths of a second, is set by the value of T.

In addition to the steps, the whole envelope is divided into three pitch sections made up of several steps. In each of these sections the pitch can rise or fall. The rise or fall of pitch for each step is defined by PC1, 2, and 3 for each section. The number of steps in each section, and thereby its length, is defined by PN1, 2, and 3.

We can illustrate this with a simple visual and aural example.

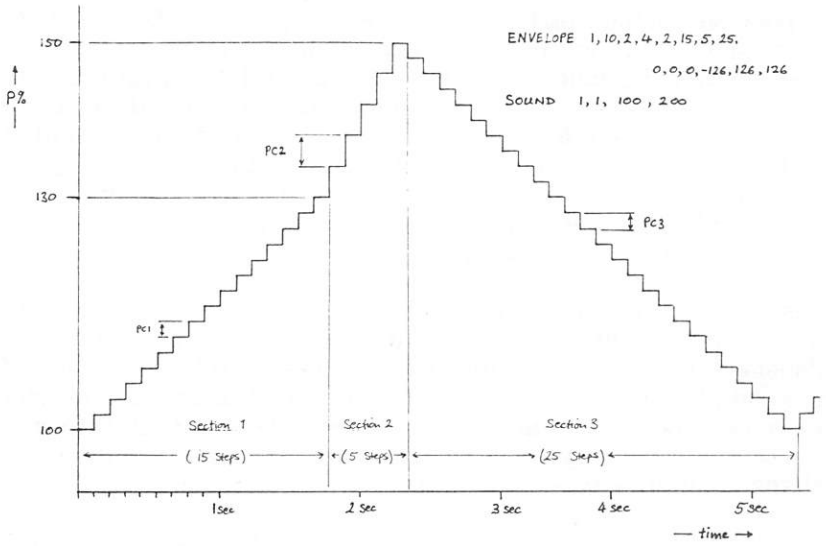


Fig. 6.4

```
   10 ENVELOPE 1,10,2,2,-2,10,10,20,0,0,0,
-126,126,126
   20 SOUND 1,1,100,200
   30 END
```

The program simply sets up an envelope and then plays a long note using that envelope. Don't worry about the last six numbers in the ENVELOPE command, we'll come on to those soon.

The diagram shows how the pitch of the note changes as the note is played. RUN the program and follow the pitch on the diagram as it rises and falls. The diagram only shows a single rise and fall. This is all that is directly defined by the ENVELOPE command. However, when the sound chip reaches the end of the pitch envelope, but there is still some note left to play according to the SOUND command's duration number, it just repeats the envelope as often as necessary to fill up the time.

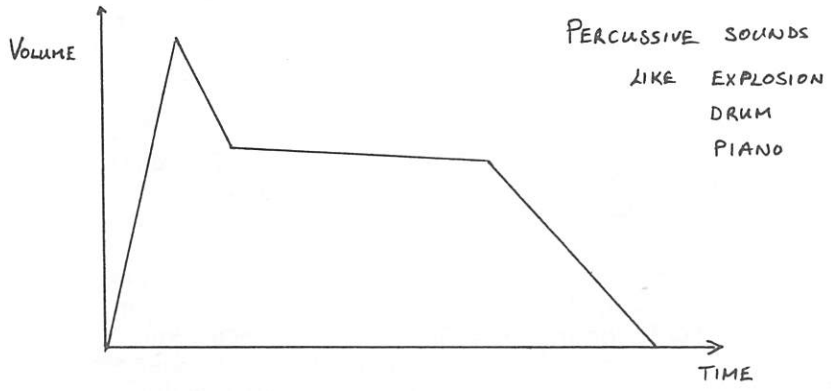
This 'auto-repeat' can be switched off by adding 128 to the value of the step length (T). Try it with this ENVELOPE and re-RUN the program. the pitch only rises and falls once now.

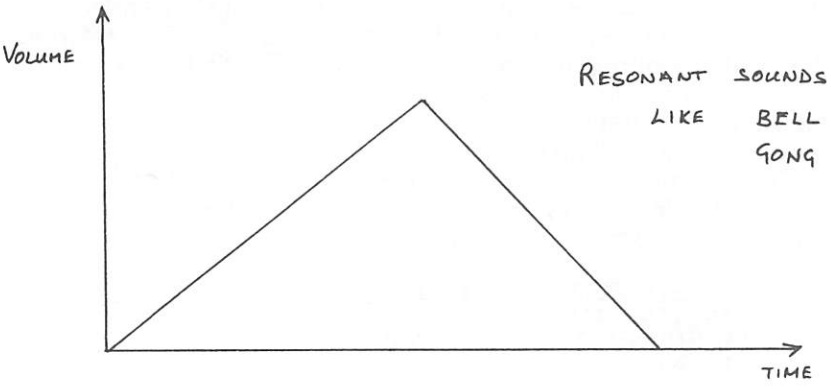
```
   10  ENVELOPE 1,138,2,2,-2,10,10,20,0,0,
0,-126,126,126
   20 SOUND 1,1,100,200
   30 END
```

We have set the step length to a pretty large value (T=10 or 138). This slowed down the changes enough to be audible, to demonstrate the action of the envelope control. However this kind of step length is not normally the most useful. Change the value of T to 1 hundredth of a second to see (or rather to hear) the kind of weird sound effects that can be obtained with BBC Basic.

So far we have only looked at the pitch changing part of the ENVELOPE command. Now, then, we'll have a go at the second half of this command, the volume part.

All natural sounds have a volume envelope. Usually this is described in the form of a sudden start, or 'attack', then a slight lessening of volume, called the 'decay' followed by the main body ('sustain') of the note and then the dying away 'release' portion of the sound. Anyone who has used a modern electronic synthesiser will be well up in this Attack-decay-sustain-release, or ADSR, format for sounds. The diagram gives you some general ideas about the shape of different types of sound.





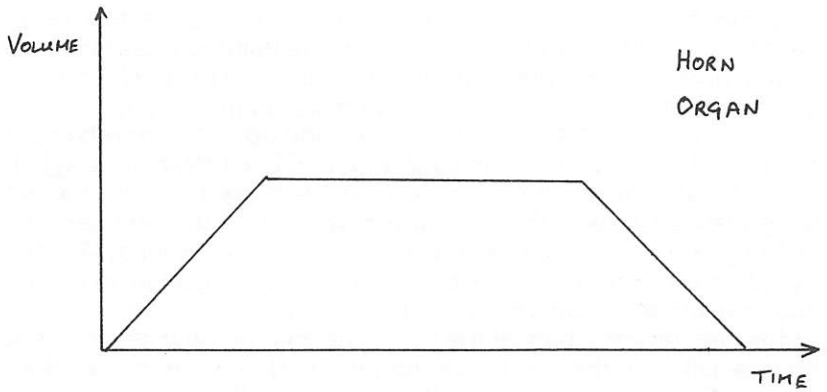


Fig. 6.5

The sound produced by a SOUND and ENVELOPE pair is split into parts for the purpose of volume control. These, however are not the same sections as are used for the pitch control above. There are four sections, called Attack, Decay, Sustain, and Release, making up the whole length of the note. There is no auto-repeat facility with the volume envelope. Once a note has finally died away it is dead for ever.

There are six numbers in the ENVELOPE command to control the volume envelope of the sound. These are defined like this:

AC	−127 to 127	Change of volume per step in A phase
DC	−127 to 127	Change of volume per step in D phase
SC	−127 to 0	Change of volume per step in S phase
RC	−127 to 0	Change of volume per step in R phase
AT	0 to 126	Target volume at end of A phase
DT	0 to 126	Target volume at end of D phase

Fig. 6.6

Like the pitch envelope, the volume envelope is based on steps of a length defined by the number T. The range of volume, however is no longer 0 to −15 as it was when just the SOUND statement was

used but from 0 to 126. 126 is equivalent in volume to –15. In addition the number of steps in each of the volume phases is also not just given by a simple number as the length of the pitch sections are. Instead there is a bit of complicated calculating to do.

The lengths of the A, D, and S phases add up to the length of the duration of the note as given by D in the SOUND statement. When the note starts the volume is increased or decreased from its present value (usually zero at the start of a new note) at the rate per step given by AC until the volume reaches the target level set by AT. The D phase starts and the note is changed in volume at a rate of DC per step until the decay target level is reached (DT).

Then we have the bulk of the note. The volume now decreases at the rate, DC until the end of the duration period. If there is another note waiting to be played, it will cut in now. Otherwise the note is released with the volume decreasing to zero at the rate RC.

If all that sounds very complicated to you, then don't worry it does to everyone else too! To help you understand it we'll have a look at another example. To keep things (fairly) simple we will keep the pitch envelope constant by making all the pitch change numbers zero.

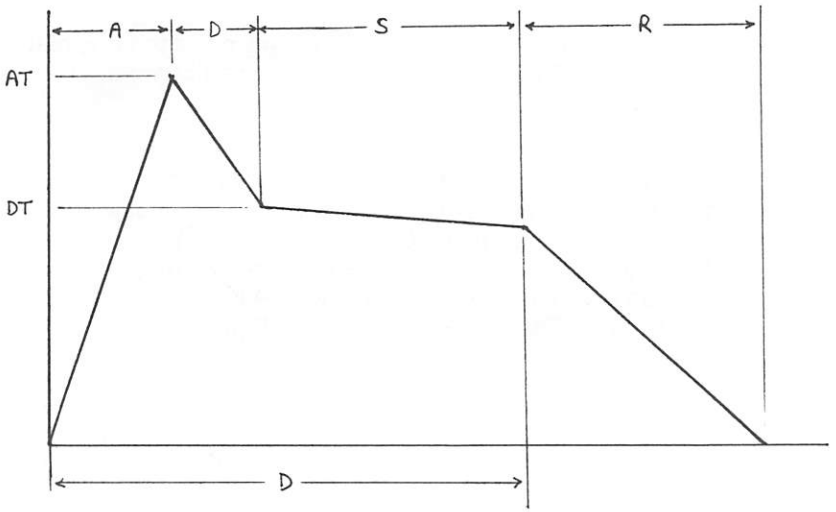


Fig. 6.7

```
   10 ENVELOPE 1,10,0,0,0,1,1,1,15,-4,-1,-
2,126,120
   20 SOUND 1,1,100,200
   30 END
```

See how the note is now a little more like you'd expect from real sound. We can get some pretty amazing sounds by using both the pitch and volume envelopes at once. First lets combine the envelopes that we have used so far:

```
   10 ENVELOPE 1,1,2,2,-2,10,10,20,15,-4,-
1,-2,126,120
   20 SOUND 1,1,100,30
   30 END
```

If that isn't good enough for you. Try these other ENVELOPE and SOUND commands.

```
   10 REM ********FIRST SOUND
   20 ENVELOPE1,1,0,0,0,1,1,1,127,-1,-1,-1
,126,110
   30 SOUND0,1,6,20
   40 SOUND1,1,10,20
   50 T%=TIME+500:REPEAT UNTIL TIME>T%
   60 REM ********SECOND SOUND
   70 ENVELOPE 1,1,20,-15,-15,6,3,3,127,0,
-1,-1,126,126
   80 SOUND 1,1,100,20
   90 T%=TIME+500:REPEAT UNTIL TIME>T%
  100 REM ********THIRD SOUND
  110 ENVELOPE 1,1,5,-5,-5,50,25,25,0,0,0,
1,126,126
  120 SOUND1,1,0,80
  130 SOUND 0,1,7,80
  140 T%=TIME+500:REPEAT UNTIL TIME>T%
  150 REM ********FORTH SOUND
  160 ENVELOPE1,1,0,0,0,1,1,1,1,0,0,-1,126
,126
  170 SOUND1,1,10,100
  180 SOUND0,1,2,100
  190 END
```

We have now covered all of the sound facilities available on the BBC micro. There is no doubt about it that the commands required to use these facilities are very complicated. There is, however, no way around that if you want to make the most of them.

As a last fling in this chapter on sound we will write a program to turn your BBC's keyboard into a musical instrument. This isn't a complicated program but it is quite a long one. We'll build it up slowly. It also contains a few (more) new Basic keywords.

What we are going to do in this program is to make your BBC micro's keyboard act as a musical keyboard to produce musical sounds. To do this we need to introduce a new keyword that detects a key being pressed. We could use INPUT for this, but having to press <RETURN> after every note would be a bit of a constraint on your style!

The keyword we are to use is GET. This comes in two forms, GET and GET$. Both are used to assign a value to a variable. Like this

```
G=GET
G$=GET$
```

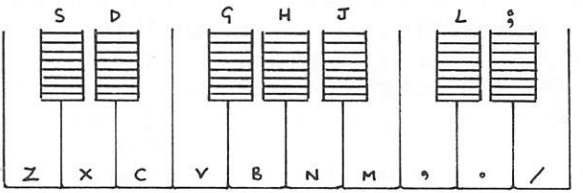
GET will wait for you to press a key and when you do it will return the ASCII code of the pressed key as the value for the variable. GET$ works in the same way but returns a string that consists of the symbol of the key pressed. GET is often used just to wait in a program until the user is ready. If you see 'Press any key to start' in a game, you can bet that GET is used there. In this program we're going to use GET$.

We want the program to detect a key being pressed and to produce a note again and again. So we will use a REPEAT-UNTIL loop. Once we know which key is pressed we want to know what note that key should produce, what pitch to use. To do this we will create a string containing all the keys that we are going to use and then search in this string for the detected key. We use INSTR for this. Remember INSTR? We met this command back in chapter three.

Type in this program as we have it so far.

```
10 KEY$="ZSXDCVGBHNJM,L.;/"
20 REPEAT
30    I$=GET$
40    N%=INSTR(KEY$,I$)
50    PITCH%=49+4*N%
60    SOUND &0011,-15,PITCH%,5
70    UNTIL FALSE
80 END
```

Line 10 sets the string, KEY$ to all the keys that we are using. These are the keys in the bottom two rows of the keyboard. We are not going to use them all. Just enough to give a representation of a piano keyboard, like this:



Line 30 waits for a key to be pressed and line 40 finds the position of that character in the KEY$. Line 50 just works out the pitch number needed for that note. Each note used is one semitone apart so we multiply the key's position in KEY$ by four to get the pitch number. We also add on 49 to the value so that the lowest key (Z) gives us a middle C. Line 60 actually produces the sound.

RUN this program and you'll find that it does the job. Play any tune you like on the 'musical' keys. You'll have to press <ESCAPE> to stop this program. However if you press one of the other keys, you'll find that they all give one unwanted, low note. This is because those keys are not in KEY$ so the INSTR returns a zero. This gives a pitch number of 49 from line 50. To get around this hitch we'll add a jump around the SOUND statement for when a wrong key is pressed.

```
55 IF N%=0 THEN GOTO 70
```

That's better. You can now play on any keys that you want but only the correct ones will produce a note.

So far so good. The quality of the notes is a bit lacking, though. What we need is ENVELOPE. Why stop at a single sound? Lets make the type of sound selectable like stops on an organ. We'll use the number keys to select what kind of envelope to use. We can do this quite easily. We already have the program lines to GET the numbers from the keyboard. All we need to do is to make a selection on the strength of these.

We'll do this with a kind of subroutine. Although we weren't going to use subroutines again, we have to here because there is no equivalent using procedures. Instead of GOSUB we are going to use ON GOSUB. This command chooses between several subroutines. Change your program to look like this.

```
    5 GOSUB 140
   10 KEY$="ZSXDCVGBHNJM,L.;/"
   20 REPEAT
   30   I$=GET$
   40   REM   Envelope selection
   50   V%=VAL(I$)
   60   ON V% GOSUB 140,160,180,200 ELSE G
OSUB 220
   70   N%=INSTR(KEY$,I$)
   80   PITCH%=49+4*N%
   90   IF N%=0 THEN GOTO 110
  100   SOUND &0011,1,PITCH%,5
  110   UNTIL FALSE
  120 END
  130 REM   Envelopes
  140 ENVELOPE 1,1,0,0,0,1,1,1,100,-20,0,-
4,126,100
  150 RETURN
```

```
  160 ENVELOPE 1,1,0,0,0,1,1,1,5,0,0,-4,12
6,126
  170 RETURN
  180 ENVELOPE 1,1,30,30,30,1,1,1,100,0,0,
-4,126,126
  190 RETURN
  200 ENVELOPE 1,1,7,7,7,3,3,3,15,-4,-1,-2
,126,120
  210 RETURN
  220 RETURN
```

The main additions are the subroutines in lines 130 to 220 and the ON GOSUB line 60. Line 50 converts the string, I$, to its value. If I$ is not a number (if you're playing a note rather than selecting an envelope) then VAL$ returns a zero. Then line 60 comes into action.

IF V% is one then the program will branch to the first line number mentioned (140). If it is two then the subroutine at line 160 is visited. And so on.

If however V% is zero (because you are playing a note, then the ELSE at the end of the ON GOSUB sends the program to the subroutine at line 220. This is just a RETURN, so the program continues on unaltered. There is no equivalent command to ON GOSUB for procedures, so we have had to use subroutines this time.

The different subroutines all work on the same principle. Each has only one line in it. This just sets a new envelope. There is a GOSUB at the beginning of the program (line 5) to initially set the first envelope. Now as you are playing the envelope can be changed by simply pressing one of keys 1 to 4.

The envelopes in this program are just examples. Change these if you want for other types. Alternatively you could add further envelopes by adding more subroutines. If you do this, don't forget to add the new line numbers of your new subroutines into the ON GOSUB statement and to change the line number of the ELSE part of this line.

Okay, now the program is beginning to shape up. All we have so far is a sound program. Let's add a bit of simple graphics. Renumber your program, then change it to look like this:

```
  3 REM   Set up screen
  4 MODE2
  5 COLOUR135:CLS
 10 GOSUB 150
 20 KEY$="ZSXDCVGBHNJM,L.;/"
 30 REPEAT
 40   I$=GET$
 50   REM   Envelope selection
```

```
   60    V%=VAL(I$)
   70    ON V% GOSUB 150,170,190,210 ELSE G
OSUB 230
   80    N%=INSTR(KEY$,I$)
   90    PITCH%=49+4*N%
  100    IF N%=0 THEN GOTO 120
  110    SOUND &0011,1,PITCH%,5
  112    REM Call to prcedure
  115    PROCPIC(N%,I$)
  120    UNTIL FALSE
  130 END
  140 REM   Envelopes
  150 ENVELOPE 1,1,0,0,0,1,1,1,100,-20,0,-
4,126,100
  160 RETURN
  170 ENVELOPE 1,1,0,0,0,1,1,1,5,0,0,-4,12
6,126
  180 RETURN
  190 ENVELOPE 1,1,30,30,30,1,1,1,100,0,0,
-4,126,126
  200 RETURN
  210 ENVELOPE 1,1,7,7,7,3,3,3,15,-4,-1,-2
,126,120
  220 RETURN
  230 RETURN
  260 REM   Procedure to print characters
  300 DEF PROCPIC(pos%,C$)
  310 COLOUR 1
  320 PRINT TAB(1+pos%,25-pos%);C$
  330 T%=TIME+25:REPEAT UNTIL TIME>T%
  340 COLOUR 7
  350 PRINT TAB(1+pos%,25-pos%);C$
  360 ENDPROC
```

Yes, we've added a procedure. This is called from line 115, so it is called every time around the REPEAT-UNTIL loop. Each time the values of the variables, N% and I$ are passed to the procedure.

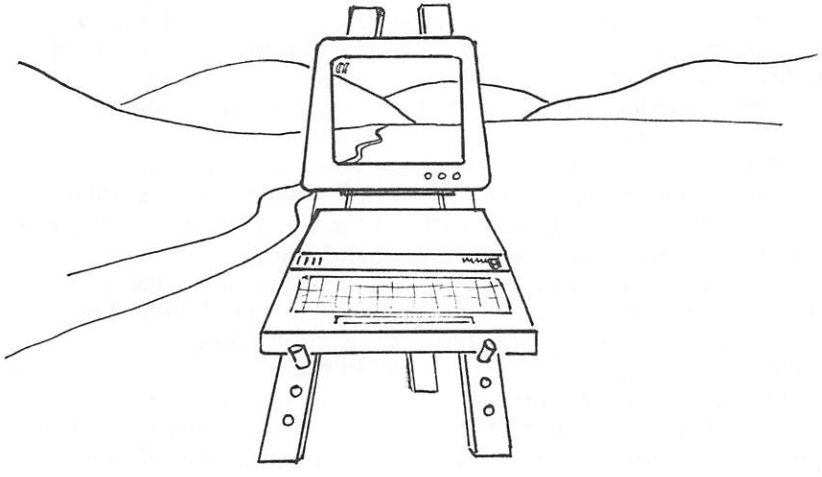
The procedure is quite simple. All that it does is to printout the key that you have pressed, in red, then wait for a quarter of a second and print it again in white to remove it from view. This removes it because the screen has been turned white all over back in line 5.

The letters are not just flashed up on the screen, but are put in a position across the screen dependent on the note that they represent. The higher the note, the further toward the top right corner goes the letter. This is accomplished with the TAB in the PRINT statement. How it is done is up to you to figure out for yourself!

There are plenty of other things that we could add to your musical keyboard program. Like selection of the range of pitch. Or playing chords. However you'll have to think up these changes for youself. We are going to leave sound now and go back to graphics again.

Chapter Seven

More graphics



In this chapter we are going to have another look at graphics on your BBC micro. We've already seen how to draw simple lines and shapes and how to fill them in. Now we shall see how to handle more complex graphics.

We have already met the idea of ASCII codes used in CHR$ commands in PRINT statements. In fact there is another way of sending these codes to the display circuitry for processing. Rather than using PRINT CHR$ (code) we can simply write VDU code.

Now, as far as producing letters or simple control codes this method does save a bit of typing, but apart from that it is not really much different to using CHR$. However there are, in the graphics modes (0, 1, 2, 4, and 5) a lot more control codes than we have met so far. A lot of these are exclusively sent to the screen using VDU.

In case you're wondering, VDU stands for Visual Display Unit, these commands are just a method of sending commands to the visual display unit of you computer. This of course is the TV screen and all the associated circuitry.

You'll find a full list of the various VDU control codes in the User Guide. Open that now at page 378. There's the list. There is also a brief description of every code on the following pages.

Some of the codes you may find familiar. This is because they have equivalent Basic commands. VDU 17 for example allows you to define the current text colour. This is exactly the same as the Basic keyword, COLOUR. VDU 25 is the same as PLOT, and VDU 12 the same as CLS.

The VDU commands are all part of the Operating system (remember that). Basic uses the operating system to manipulate the screen. In Basic you can do this, in the case of some actions, either by using the Basic commands or the relevant VDU commands. On the whole it is better to stick with the Basic keywords where possible because they are more friendly and easy to remember. Some of the actions have no equivalent Basic keyword so we are forced to use the rather horrible VDU format. It is these commands that we shall look at in this chapter.

VDU codes 7 to 11 we have already met in their guise as ASCII codes, used with CHR$. VDU 13 is another that is a bit similar. It is equivalent to a <RETURN> without the movement onto the next line. Things start to get interesting at code 16.

You can see that this is used to 'clear the graphics area'. Now this is a bit strange as we already have a command that clears the screen, CLS or VDU 12. So what's this graphics area?

Normally the graphics area is indeed the entire screen, as is the text area, so clearing either has the same effect. The whole screen is cleared. However we can set aside small areas of the screen (called windows) as areas just for text or just for graphics. This is done

using VDU codes 24 and 28. This short program illustrates this idea.

```
  5 REM graphics windows
 10 MODE 2
 20 C%=1
 30 FOR X%=1000 TO 100 STEP -150
 40   left%=(1280-X%)/2
 50   right%=1280-left%
 60   bottom%=(1024-X%)/2
 70   top%=1024-bottom%
 80   VDU 24,left%;bottom%;right%;top%;
 90   GCOL 0,128+C%:C%=C%+1
100   CLG
110   NEXT X%
120 T%=TIME+100:REPEAT UNTIL TIME>T%
125 REM text window
130 VDU 28,2,22,6,10
140 END
```

The graphics windows part of the program defines a series of windows, each one smaller than the last and in a different colour. This is done with a FOR-NEXT loop using a negative step size so that the loop variable decreases in value. Lines 40 to 70 calculate the graphics co-ordinates of the four sides of the rectangular windows each time around the loop. Then line 80 actually defines the window using these values.

The window is defined with four values. These are the x co-ordinate of the left side, the y co-ordinate of the bottom, the x co-ordinate of the right side and the y co-ordinate of the top.

These must be in this order and they must be each followed with a semicolon in the VDU 24 statement.

Line 90 selects a new background colour for each window and line 100 clears the graphics window with this colour. The net result of all this is a series of coloured rectangles, one on top of the other.

The program waits for one second after all the graphics windows have been defined and filled, and then a text window is defined in line 130. The order of defining the sides is the same for text windows as it is for graphics windows but the co-ordinates used are the character position co-ordinates used for the TAB command. Don't forget that these have the origin at the top left and not the bottom left as the graphics co-ordinate system has.

This text window is not cleared to take any particular colour, so you cannot see it there. However you can see the prompt (>) and the cursor appear in the top left hand corner of the window. The cursor is always moved to the home position of any newly defined window.

The action of the text window will become immediately apparent if you now LIST the program. Text only appears in the text window.

The line length is all taken care of by the micro so that you effectively have a small separate text screen in the centre of your TV screen.

The effect of the graphics window is similar. Graphics will only show inside the graphics window. This is why the areas are called windows. It is as though you are looking through a window at a larger graphics screen. Type in this at the keyboard and you'll see this windowing in action.

```
GCOL 0,0
MOVE 0,0
DRAW 640,512
```

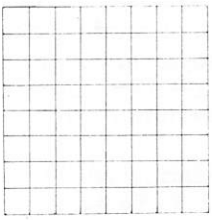
If the graphics window was over the entire screen, then these commands would draw a black line from the bottom left hand corner to the centre of the screen. However you can see that only that part of the line that is inside the last graphics window appears. Both the text and graphics windows can be restored to their default sizes and positions, on the entire screen with the command VDU 26.

Windows are very useful for programs that require a lot of text to appear on the screen, causing it to scroll, without disturbing a picture there. You could see the effect of scrolling text on one bit of the 'picture', created by all our graphics windows, inside the text window.

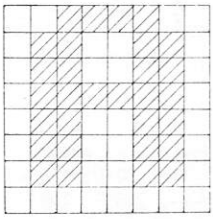
Windows are not the only graphics effect that can be created using VDU codes. Another, most important one is not really a graphics effect at all, but a text one. However it is most usually used with graphics. We are now going to have a look at 'user definable characters'.

The definition of VDU 23, you will see from your User Guide, is to 'Re-define a display character'. There are a large number of characters available on the screen from your BBC micro. 95, in fact, in all the non-teletext modes. However none of these are very useful for anything but writing text. That is what they are there for. There are no space invader characters for example, nor letters of the Greek alphabet. It is possible though to take matters into your own hands and solve this problem by creating you own characters.

All the characters already defined in your BBC micro are based around an 8 by 8 grid of dots. Like this



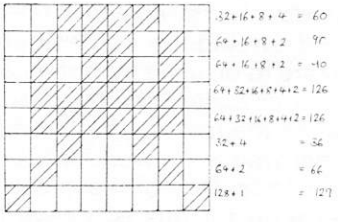
Each cell in this grid is either 'lit' or 'unlit' in the correct pattern to produce a character. The letter A for example is formed like this



VDU 23 is a command that allows you to design your own characters on this grid. Any of the characters with ASCII codes of 224 to 255 can be defined in this way. These characters do not otherwise have any design assigned to them. You can in fact re-design any of the characters, including those that normally have a symbol already attached to them, but this requires you to set aside memory in the computer for this and so it is not recommended for beginners. At any rate 32 of your own characters should be enough for most purposes.

VDU 23 commands take the form of VDU 23 followed by the code of the character you want to define, and then eight numbers. These numbers are the design itself. Each represents one row of the 8 by 8 grid. The way in which these numbers are found is rather similar to the way in which you found the ASCII code for the graphics characters in mode 7.

Each column of the grid is given a number. These are, counting from the left, 1, 2, 4, 8, 16, 32, 64, 128, each double the last. Using these numbers any arrangement in a row can be represented with a unique number between 0 and 255. Once you have designed your character on a grid you simply add up all the lit column numbers for each row like this:



VDU 23, 224, 60, 90, 90, 126, 126, 36, 66, 129

There we have designed a space invader character. If you type in this VDU 23 command, from then on printing character number 224 will produce this alien on the screen. Your micro will only 'forget' the design if you re-design that character, or if you give you machine a 'hard' reset (<CTRL> and <BREAK>) or if you switch it off.

RUN this next short program to demonstrate user definable characters.

```
  5 REM define characters
 10 VDU23,224,60,90,90,126,126,36,66,129
 20 VDU23,225,0,60,102,102,102,36,36,66
 30 VDU23,226,1,3,7,8,8,8,8,8
 40 VDU23,227,128,192,224,16,16,16,16,16
 50 VDU23,228,8,24,40,72,127,2,4,7
 60 VDU23,229,16,24,20,18,254,64,32,224
 65 REM print out characters in each mod

 70 FOR M%=0 TO 6
 80   MODE M%
 90   PRINT CHR$(224);CHR$(32);CHR$(225)
100   PRINT
110   VDU226,227,10,8,8,228,229
120   T%=TIME+100:REPEAT UNTIL TIME>T%
130   NEXT M%
140 END
```

The first 6 lines uses VDU 23 commands to define various characters. The second half of the program then prints out these characters in all the non-teletext modes. The first two characters (the alien and a Greek omega) are printed using PRINT CHR$ along with a space character (32) to separate them.

The next four are printed using just VDU as we mentioned earlier. These four are designed as a super sized character, made up of four characters. The cursor control codes, 8 and 10, are used to position these four characters in a block.

You can see how the characters change their proportions in the various modes. This is a result of the different numbers of characters that make up the screen in the different modes. This can't be avoided and should be remembered if you are designing characters that are to be used in modes 2 and 5 (the oblong modes) especially. In modes 3 and 6 the big character is rather spoilt by the gaps that separate the rows in these modes.

There are other weird and wonderful effects that you can achieve with VDU commands. One of these is to shift the graphics origin. The default position for this is, as we have already seen, at the bottom left hand corner of the screen. Using the VDU 29 command the origin can be defined as any point on or off the screen. The new

origin is given as a set of co-ordinates based on the default origin. Both of these co-ordinates, x and y, must again be followed by a semicolon.

The greatest use for this origin moving is when drawing the same shape again and again in different places around the screen. Rather than spending time and program lines calculating the co-ordinates for the various points of an object in all its different positions on the screen, you calculate the co-ordinates just once, around a central reference point. Then for each position that you want the object to appear in, you just redefine the origin to that position, and draw your object with relative co-ordinates around that point.

We shall be using this method in a program later in this chapter.

There is another VDU code that we should include at this point. So far we have kept a firm distinction between graphics and text. Their cursors are different, their colour commands are different, their windows are different, and so on. However there is a limit to the usefulness of text characters, and more importantly to user-defined characters, when they can only be placed in the limited number of positions used by the TAB command. What we really need is a means of positioning text to the same accuracy as we can position graphics points. Fortunately that is exactly what VDU 5 enables us to do. If you type in VDU 5, or include this in a program, from then on all text is printed at the position of the graphics cursor. This means that

```
MODE 1
COLOUR 3
PRINT TAB(10,5)"Hello"
```

can be replaced by

```
MODE 1
VDU 5
GCOL 0,3
MOVE 320,864
PRINT"Hello"
```

or for that matter any MOVE statement.

Now that your text is effectively graphics it appears in the graphics window and its colour is set by the GCOL command and not the COLOUR command. The price that you have to pay for this versatility is a reduction in printing speed. When the text and graphics cursors are joined in this way (that is what this process is called) characters are sent to the screen at a slightly slower speed. The speed is still pretty good, so this slowness will not effect a lot of applications.

The final VDU code that we are going to look at at this time is number 19. When we first introduced the different display modes,

we saw how the different modes could produce different ranges of colours. Mode 2 could produce all sixteen and the other modes only two or four colours. In fact this is not really true. All the modes can produce all eight colours. What differs between modes is the number of colours that can be displayed at any one time.

Mode two can indeed display all sixteen colours and mode 1, for example, can only display four, but those four can be picked from any of the sixteen. In fact when we say COLOUR 1 or GCOL 0,1 we are not selecting red but colour number one. The default colour for number one is indeed red, but this can be changed.

The colours that we refer to in colour commands are known as 'logical' colours. The actual colours that appear on your TV screen are called (not too surprisingly) 'actual' colours. We can define which actual colour appears for any logical colour using the VDU 19 command. Normally the actual colours displayed for each of the logical colours in the various modes are as given in the table in chapter 5, but these can be changed.

If you've been following this chapter properly you should now be in mode 1. That is the mode that the last program left you in. Your display consists of white text on a black background. Type in this

```
VDU 19,0,2;0;
```

Hey presto! The background has gone green. Now try this

```
VDU 19,3,1;0;
```

Now the text has gone red. Notice that after both of these commands the whole screen changed colour. It is not just a case of changing the current colour so that all text from then on is different. What we have done is to redefine black as green and white as red.

This can be done for any colour. We can redefine any of the logical colours available in any mode to appear in a different guise as a different actual colour. The syntax of the VDU 19 statement is very simple. It goes like this

VDU 19,L,A;0;

L is the logical colour number that you wish to change and A the actual colour number that you want it to appear as. Note the zero, commas and semicolons. these must be included as given here. The number codes of the actual colours are those default values for the logical colours in mode two. We've already seen these, but to remind you

0	Black	8	Black/White
1	Red	9	Red/Green
2	Green	10	Green/Magenta
3	Yellow	11	Yellow/Blue
4	Blue	12	Blue/Yellow
5	Magenta	13	Magenta/Green
6	Cyan	14	Cyan/Red
7	White	15	White/Black

What we have just done is to define the logical colour zero (which is the default colour of the background) to actual colour 2 (green) and to define logical colour 3 (the default colour for the foreground) to actual colour 1 (red).

Put some more red onto the screen by typing in this short list of immediate commands.

```
COLOUR 1
PRINT "red text"
COLOUR 3
PRINT "white text redefined as red"
```

Both pieces of text appear in red. Now redefine colour 3 again

```
VDU 19,3,8;0;
```

Only the second of the two pieces has changed to colour 8 (White/Black) the other is still red. This is because it is a different logical colour and so was not effected by your redefining of logical colour 3. You can also see how **all** the text in logical colour 3 has changed. Not just the last piece.

Remember how, in chapter 5, we saw 'all' the colours available in each mode using a program that changed the colour of the background? Now we can write a similar program that will display all sixteen colours in mode zero — what we thought was a two colour mode.

```
10 MODE 0
20 FOR C%=0 TO 15
30   VDU 19,0,C%;0;
40   T%=TIME+150:REPEAT UNTIL TIME>T%
50   NEXT C%
60 END
```

So now we can put any of the eight steady and eight flashing colours onto any display mode screen. The actual colours can be easily reset to their default values by either using VDU 20 or by changing the screen mode. Try it.

There was another command that we only glossed over in chapter five. If you remember, we change the current graphics colour with the command, GCOL. So far we have only used one form of this. Thee are in fact five forms of GCOL numbered zero to four. The number denoting the form you are using comes immediately after the GCOL command. So far we have only used GCOL 0.

The five forms are

GCOL 0	Overwrite with the colour
GCOL 1	OR the colour with that already there
GCOL 2	AND the colour with that already there
GCOL 3	EOR the colour with that already there
GCOL 4	Invert the colour already there

GCOL 0 is the simplest. It causes the colour you chose (say red in mode 2 with GCOL 0,1) to be put on the screen so that it covers whatever is alredy there.

It is as though you had painted over the screen with a brush dipped in red paint.

GCOL 4 is almost as easy to understand. This simply causes the inverse of what ever colour is already there to appear. We have already seen a similar effect with the PLOT 2 and PLOT 5 commands.

The other GCOL types are more complicated, OR, AND and EOR are mathematical operations. What these forms of GCOL do is to perform these operations between the logical colour that you choose and that colour already on the screen at that point, and put the result onto the screen.

It is really beyond the scope of this book, as a guide for beginners, to go into the details of these operations but we are going to take a look at a use of the GCOL 3 form. Run this simple program.

```
  10 MODE 1
  20 COLOUR 130:CLS
  30 REM overwrite
  40 GCOL 0,3
  50 PROCSQUARE(900)
  60 REM inverse
  70 GCOL 4,3
  80 PROCSQUARE(600)
  90 REM use EOR
 100 GCOL 3,3
 110 PROCSQUARE(300)
 120 END
 130 REM draw square with top at Y%
 140 DEF PROCSQUARE(Y%)
 150 MOVE 100,Y%:MOVE 300,Y%
 160 PLOT 85,100,Y%-200:PLOT 85,300,Y%-20
0
 170 ENDPROC
```

This program draws three rectangles on a yellow background. Each square is drawn by the procedure, PROCSCQUARE at a point on the screen given by Y%. Each square is drawn with a logical colour of three (normally, and still unchanged from white) chosen in the GCOL command.

The top square is white because the overwrite form of GCOL, GCOL 0, was used to set the current graphics colour to three. The middle square is Red because GCOL 4 gives logical colour 1 (red) which is the inverse of the logical colour 2 (yellow) already there.

The third rectangle, however, is also red. This is because 3 EOR 2 (our chosen logical colour EOR'ed with the colour already there) is 1 (red). Type PRINT 3 EOR 2 to prove this. Both GCOL 3 and 4 (and 1 and 2 for that matter) tend to produce weird results when they are used on areas next to one another. This is because the areas actually overlap. That is why, in the second and third squares, there is a diagonal line where the two triangles that make up the square overlap. The real use of GCOL 3, however, comes when you do things twice. Type in this

```
GCOL 3,3
PROCSQUARE(300)
```

You can call procedures in immediate mode, and that is what we have done. We have called PROCSQUARE just as the program did. The bottom square has disappeared. This is because 3 EOR 1 is 2, the background colour. Call the procedure again and the square will appear again. Call it once more and it will vanish. And so on for ever. We can use this for animation.

The next program gives you a taste of this.

```
  5 REM define character
 10 VDU23,224,60,90,90,126,126,36,66,129
 15 REM set up
 20 MODE 2
 40 VDU 5
 50 GCOL 3,2
 60 X%=RND(1000)+100
 70 Y%=RND(800)+100
 80 REPEAT
 85    REM print character
 90    MOVE X%,Y%
100    VDU 224
105    REM calc. new position
110    X%=RND(1000)+100
120    Y%=RND(800)+100
125    REM wait
130    T%=TIME+20:REPEAT UNTIL TIME>T%
135    REM go back and print character
136    REM again to erase it
140    VDU 8,224
```

```
145    REM loop around again
150    UNTIL FALSE
160 END
```

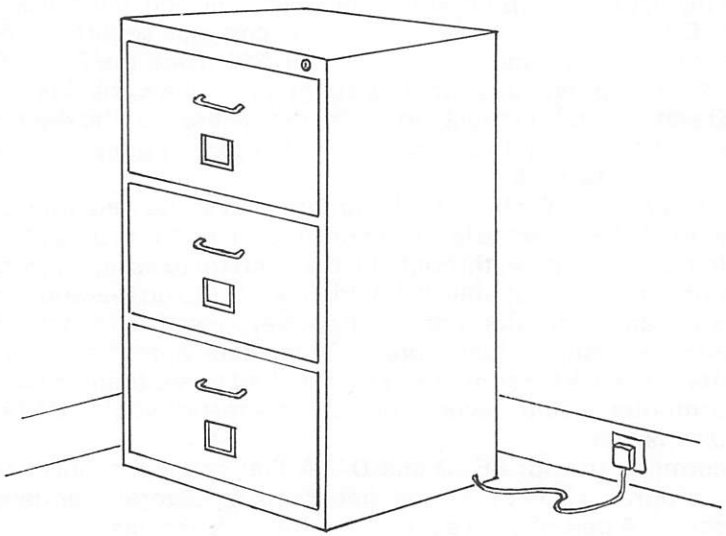
The REM statements explain the workings of the program. Notice how all the time-consuming parts of the program are between successive printings of the character in the same position and not between positions. This means that the character spends more time still than moving so that the animation is smoother and more flicker-free. You'll have to press <ESCAPE> when you're through with this one.

We have now covered just about every aspect of graphics on your BBC micro. The most tempting and complex use of the graphics, animation is the subject of the next chapter but one. First though, we will take a look at data.

It is data that makes the modern world go round. It is data that computers are really at home with. So then it is methods of dealing with data that we look at in the next chapter.

Chapter Eight

Data-day programs



We are going to take a break from the joys of BBC graphics in this chapter to look at the treatment of data by your micro. Everything that your BBC can do it does by manipulating numbers. Graphics and sound are just so many numbers to your micro. In this chapter we are going to see how you can use your micro to manipulate your own numbers.

So far in this book, whenever we have wanted to assign a number or a string to a variable we have either just assigned the value with a statement such as X=47 or we have left the value up to the user at the time by using an INPUT statement such as INPUT X. There is however a further way of assigning numbers and strings to variables. We can store all of the values in a special part of the program called DATA statements. These values are then assigned to the variables with READ statements. This short program will show you how this works.

```
   10 DIM M$(11),N%(11)
   20 FOR I%=0 TO 11
   30   READ M$(I%),N%(I%)
   40   PRINT M$(I%);" is month number ";N
%(I%)
   50   NEXT I%
   60 END
   70 DATA JANUARY,1,FEBRUARY,2,MARCH,3,AP
RIL,4,MAY,5,JUNE,6,JULY,7,AUGUST,8,SEPTEMB
ER,9,OCTOBER,10,NOVEMBER,11,DECEMBER,12
```

All the data for this program is stored after the end of the program in the DATA statements. Pay note to the commas separating each piece of data. You must use these. Line 30, inside the FOR-NEXT loop, takes the data in pairs and assigns it to the variables in the READ statement. You must, however, make sure that the right data ends up with the right variable. The micro cannot guess what you mean, it must be told.

In this program we have both numeric and string data (numbers and words). If we were to leave out one month number, the program would merrily carry on through the data and try to assign a word to the month number variable. It wouldn't be able to do this and would generate an error. Sometimes, however, you might have just numbers as data or just words. If two data numbers were for example switched around, because you had typed them in wrong, the computer would never know but the result would not be as you'd expected.

A common use for READ and DATA that you'll probably see in many programs is in programs that are using several user defined characters. A part of such a program may look like this.

```
100 FOR N=224 TO 255
110   READ A,B,C,D,E,F,G,H
120   VDU 23,N,A,B,C,D,E,F,G,H
130   NEXT N
```

At the end of the program there should be all the groups of eight numbers defining the characters in DATA statements. Please note that this is *not* a complete working program.

It is good practice to keep all your DATA statements at the end of a program, after the END statement. The data can, however, come at any point you wish. There is, hidden away in your BBC's memory, a sort of pointer that points to the next piece of data to be read. When the program comes across a READ statement it reads the data at the pointer and then moves the pointer on to the next piece of data. That next piece might be in the same line or it might be in a totally different part of the program. All that matters is that it is the next piece following a DATA statement.

What this means is that as soon as you've read in your data once the pointer has passed it and you can't access it again. That is you can't unless you use a RESTORE command. Add these lines to your month program.

```
15 FOR GOES=1 TO 2
55   NEXT GOES
```

Now you are asking the micro to print out the whole list twice. When you RUN this program you will find that the computer comes back with an error, 'Out of data' after the first run through the list. The data pointer has run off the end of the data list.

We can, of course, add program lines containing the same DATA statements all over again, but this would be rather memory consuming for any large amounts of data. A much better method is to use RESTORE. This command resets the data pointer back to the first piece of data. Add in this line to the program.

```
16 RESTORE
```

Now the pointer is reset before each run through the list and the program works well.

RESTORE is in fact far more powerful than this. We can reset the pointer to any part of the list of data. If we follow the command RESTORE with a line number, the pointer will be reset to the first bit of data in the DATA statement in that line. We can see this in action in this program giving you information about different microcomputers.

```
  10 MODE 4
  20 PRINT '' "SELECT ONE OF "'"1 COLOUR"
'"2 MEMORY SIZE"'"3 KEYBOARD"
  30 REPEAT
  40   INPUT TAB(2,8)"SELECTION",N
  50   RESTORE
  60   FOR I%=0 TO 4
  70     READ NAME$
  80     PRINT TAB(2,I%+10)NAME$
  90     NEXT I%
 100   RESTORE N*10+170
 110   FOR I%=0 TO 4
 120     READ INFO$
 130     PRINT TAB(20,I%+10)INFO$
 140     NEXT I%
 150   UNTIL FALSE
 160 END
 170 DATA BBC,Spectrum,Oric,CBM 64,Memote
ch 500
 180 DATA Beige,Black,White,Grey,Black
 190 DATA 32K  ,16K   ,16K   ,64K ,32K  ,
 200 DATA Full ,Rubber,Rubber,Full,Full ,
```

Each time that you make a selection the pointer is reset to the beginning of the data and the micros' names are printed out. Then the pointer is reset to the line number of the data relevant to your selection and that information is printed out.

We are getting close to a library in your BBC. Computers are used all over the world for holding and handling information about everything from details of the weather to criminal records. A store of information held in a computer is called a 'database'. Let's write a very simple database on your BBC micro.

We will create a database on various home micros. A good database can not only access data but sort out how much of it you really want to see, here is our program. Type it in.

```
  10 DIM NAME$(4),INFO%(4)
  20 REPEAT
  30   MODE 7
  35   REM *** print out choices ***
  40   PRINT '''"Select one of:"
  50   PRINT TAB(4,5)"1";CHR$(134);"    M
emory size"
  60   PRINT TAB(4,6)"2";CHR$(134);"    N
o. of columns"
  70   PRINT TAB(4,7)"3";CHR$(134);"    N
o. of colours"
  80   PRINT TAB(4,8)"4";CHR$(134);"    N
o. of keys"
  85   REM *** input and test choice ***
  90   INPUT TAB(4,12),CHOICE%
```

```
  100    IF CHOICE%>4 OR CHOICE%<1 THEN GOT
O 30
  105    REM *** flash chosen option ***
  110    PRINT TAB(1,4+CHOICE%)CHR$(136)
  115    REM *** input and test limits ***
  120    INPUT TAB(4,12)"Give limits   UPPE
R ",ULIM%
  130    INPUT TAB(18,13)"LOWER ",LLIM%
  140    IF ULIM%<LLIM% THEN GOTO 30
  145    REM *** read in micros names ***
  150    RESTORE
  160    FOR I%=0 TO 4
  170      READ NAME$(I%)
  180      NEXT I%
  190    RESTORE CHOICE%*10+320
  195    REM *** read units of info ***
  200    READ N$
  210    L%=0
  215    REM *** read in info ***
  220    FOR I%=0 TO 4
  230      READ INFO%(I%)
  235      REM *** test info against limits
 ***
  240      IF INFO%(I%)>ULIM% OR INFO%(I%)<
LLIM% THEN GOTO 270
  245      REM *** print out name and info
if within limits ***
  250      PRINT TAB(4,18+L%)NAME$(I%) TAB(
12,18+L%)INFO%(I%);" ";N$
  260      L%=L%+1
  270      NEXT I%
  275    REM *** if none printed, say so!**
*
  280    IF L%=0 THEN PRINT TAB(5,18)"None"
  285    REM *** wait for response ***
  290    G=GET
  300    UNTIL FALSE
  310 END
  315 REM *** all the data ***
  320 DATA BBC,Spectrum,Oric,CBM 64,Memote
ch
  330 DATA K,32,16,16,64,32
  340 DATA columns,80,32,40,40,40
  350 DATA colours,8,8,8,16,16
  360 DATA keys,72,40,57,66,79
```

It's longer than anything we've done yet but that is because it does a job well. This program is to help you to choose between home micros. It is a bit late for that now, admittedly, but it serves a useful demonstration purpose. You are first presented with a 'menu' of possible criteria which you may like to use to choose between the machines. This part of the program should give you no

problems. After you have typed in a choice, that choice is checked to see if it is within the possible limits. This last process is very important and one which is often missed from programs of all natures. All good programs should guard against mis-typing and be generally 'idiot-proof'.

The selction that you have chosen is used to point to the right area of data just like the last program. This time, though, only the information that is within the limits that you specify is printed onto the screen.

In this example we are choosing between five micros only, and using only four different criteria. Obviously this task would be much easier to do 'by hand'. However the principal of the program stands for any amount of data. It is possible to change this program to choose between two hundred micros with twenty features to look at. See if you can add to the data lists at the end of the program and then change the program to take account of your new data.

The problem with expanding programs of this type, to encompass large amounts of data, is that there is only a limited memory space in your BBC micro to store it all in. In addition, adding or removing all the data for one particular subject is very difficult to do when there are many more subjects than there are micros in this program.

Instead we use a data organisation system known as 'files'. You have of course met the idea of files before in everyday life. Computer files are very similar to their real life equivalent.

One form of file system that you can implement on your BBC micro is the DATA statement system that we have just seen. There are two other systems. These use a cassette recorder or a disk drive. Disk drives are by far the best method of storing data files because they are quick to access data and they can go directly to a piece of data under computer control. Cassette recorders on the other hand are very slow at storing and releasing data. The data on a cassette can only be read in sequence as it is recorded on the cassette. However, disk drives are much more expensive than cassette recorders so we are going to concentrate on the use of tape. Most of the ideas are however common for both recording 'media' so if you are fortunate enough to have a disk drive or two, you won't feel left out.

First: What do we mean by a file? A file is any collection of data. In a file there are many records. The analogy with everyday life would be this.

A card index is a file. Each of the cards hold several pieces of information and are records. Each piece of information (like the name of the author, the subject, and the shelf number, in a library index) is called a field.

Ideally what we would like to do in our micro choosing example would be to store all the information in permanent form then just grab back one record at a time, check it to see if it falls within our limits and print it if it does. Unfortunately this requries what is known as 'random access'. That is the ability to go straight to any record in a whole file. With disks that is easy. With cassettes things are different. The computer has no control over your cassette recorder. Short of continually producing messages saying things like 'rewind the cassette back a bit' there is no way that your micro can produce random access files on cassette.

What we have to do is to use a variant of the idea that we have already explored with the DATA statements. Now, instead of storing the data in DATA statements we will store it in sequential cassette files. Although this means that we are still limited to the memory size of the micro, at least we can now easily alter the records.

The way in which files are organised on your micro is a bit odd. This is because the methods date back a long way to situations when they were necessary. However you're stuck with it.

Files are accessed through 'channels'. It is as though you are in charge of a large office with five secretaries. Each secretary can only deal with one filing cabinet at a time.

You might tell one secretary at the beginning of the day, 'Doris, you look after the customer accounts file today'. From then on through the day you would just give her files and ask her for files. You would not need to know the filing system, nor which cabinet the files are kept in, only that Doris is looking after customer accounts.

To your micro, of course, the channels are not called anything like Doris. They are numbered, but you do not actually have to know which number has been assigned to any channel. You only have to give it a name, like Doris.

If we want to store some data onto a cassette file, we have to first open the file. We do this by saying something like

```
DORIS=OPENOUT("CUSTA/C")
```

There, we have told Doris to stick a label called 'CUSTA/C' on a filing cabinet somewhere. From now on, to put an item of data into this file you just say 'Here, Doris, put this in the file, will you'. Or in Basic

```
PRINT# DORIS,data,data,data,data...
```

PRINT# is totally different from PRINT. Once we have put as much information into the file as we want to we must release Doris to get on with her other tasks. We must close the file, like this

```
CLOSE# DORIS
```

Let's set up a file called 'Micros' to hold all the information that we used in our micro choosing program.

Setting up a file like this requires a program in two parts. First the data must be put into the micro, from you, and then secondly this data must be stored on a cassette in a file form. This program will set up a file called 'Micros' containing facts and figures on micros' memories, colour capabilities and keyboards.

```
   10 MODE7
   15 REM *** open file ***
   20 DORIS=OPENOUT("Micros")
   30 REPEAT
   40    CLS
   45    REM *** input data on micros ***
   50    PRINT TAB(2,4) CHR$(134);"Name of
micro"
   60    PRINT TAB(2,5) CHR$(134)"(less tha
n 10 chr.s)";
   70    INPUT NAME$
   80    IF NAME$="F" OR NAME$="f" THEN PRO
CFINISH:END
   90    PROCCLEAR
  100    PRINTTAB(5,15)NAME$
  110    PRINT TAB(2,4) CHR$(134);"Memory s
ize"
  120    PRINT TAB(2,5)CHR$(134);"of the ";
NAME$;" (k)";
  130    INPUT MEM%
  140    PROCCLEAR
  150    PRINTTAB(15,15)MEM%;" k"
  160    PRINT TAB(2,4) CHR$(134);"No. of c
olumns"
  170    PRINT TAB(2,5) CHR$(134);"in the "
;NAME$;"'s display";
  180    INPUT COL%
  190    PROCCLEAR
  200    PRINTTAB(15,17)COL%;" columns"
  210    PRINT TAB(2,4) CHR$(134);"No. of k
eys"
  220    PRINT TAB(2,5) CHR$(134);"in the "
;NAME$;"'s keyboard";
  230    INPUT KEY%
  240    PRINTTAB(15,19)KEY%;" keys"
  245    REM *** all okey? ***
  250    PROCCLEAR
  260    PRINT TAB(2,5)CHR$(130);"Are all t
hese correct ?";
  270    ANSWER$=GET$
  280    IF ANSWER$="N" THEN GOTO 30
  285    REM *** store answers ***
  290    PROCKEEP
  300    UNTIL FALSE
```

```
  310 END
  315 REM *** clear input area of screen *
**
  320 DEFPROCCLEAR
  330 PRINT TAB(2,4) SPC(30)
  340 PRINT TAB(2,5) SPC(30)
  350 ENDPROC
  355 REM *** output micro-data to cassett
e ***
  360 DEFPROCKEEP
  370 VDU7
  380 PRINT#DORIS,NAME$,MEM%,COL%,KEY%
  390 ENDPROC
  395 REM *** close file (actually record
data) ***
  400 DEFPROCFINISH
  410 CLS
  420 PRINT TAB(10,10)CHR$(136);CHR$(131);
"PLEASE WAIT"
  430 CLOSE# DORIS
  440 CLS
  450 ENDPROC
```

It is pretty long! Fortunately it is not all that complicated. First of all the file is open in line 20. This must be done in all work like this. When this line is executed, when the program is RUN, the message 'Record then return' will appear. As soon as you have pressed record on your tape recorder and pressed <RETURN>, your recorder will start. Nothing is actually recorded at this stage, though. It is almost essential to have a recorder with a remote control facility for this program. Otherwise it will be running all the time that this program is running with only short snippets of data actually recorded along a very long blank tape.

Once the file is opened, the program gets on with inputing the data from the keyboard. This is done with a series of prompts. Lines 50 and 60, for example prompt for a micro's name, then line 70 inputs that name and stores it as NAME$. Then the name is printed at the foot of the screen in line 100. The rest of the data is similarly prompted for and stored in various variables, then printed out on the screen.

The micro's name that you have entered is displayed in the prompt for the other data. This is a very simple trick that makes the program more pleasant to use. It is always a good idea to try and include as many friendly gestures such as this is your programs.

Once all the details are in and displayed on the screen, by line 250, a check is made to see if it is all correct. It is a waste of tape to store incorrect information, so we might as well do as much checking as possible now. The user (you) is asked if everything is alright. If it is,

then the program moves on to a procedure called, PROCKEEP, otherwise the input part of the program is run again to get new values for all the data variables.

PROCKEEP is really very simple. When it is called, a beep is sounded and the data is output to the file from line 380. Nothing is necessarily actually recorded onto your tape at this instant. All data sent to a file, in this way, is just added to a queue. It is a bit like the SOUND queue that we have already met. Once the queue is full, then a whole block of data is recorded onto the tape at one time.

This means that PROCKEEP usually takes very little time. The data is usually just added to the queue in memory. However, every so often, the queue becomes full and another block is recorded. When this happens, the program will appear to cease working for a while, as the micro is busy with the cassette recorder. You will be able to tell these moments quite easily as your cassette recorder will spring to life then, as the data is recorded.

The whole inputting process (lines 30 to 300) is repeated indefinitely so that you can input the details of lots of micros for your micro database.

Once you have finished inputting the data on all the micros that you want to cover in one sitting, you can stop the process by typing F the micro's name when prompted for this. This is detected in line 80 and sends the program off to PROCFINISH.

The main thing that this procedure does is to close the file. Again this is essential in all work of this kind. In fact, if you find that this program gives you an error as it is running because you have, say, mis-typed a part of it, then you should close the file after the program has stopped. it is not a good idea to leave open files lying around!

When the file is closed with the CLOSE command in line 430, the cassette recorder will again come to life. Regardless of the length of the data queue, closing a file will send the last bit off to the cassettte recorder. This is why the 'Please wait' message is included in PROCFINISH.

This program will create a file on cassettte for you of the data on several micros. Of course now you want some way of accessing your new database, to retrieve this information, as and when you want it. This is what the next program does.

```
    5 REM *** set up ***
   10 DIM INFO$(4)
   20 N$="":ULIM%=0:LLIM%=0
   30 INFO$(2)=" k":INFO$(3)=" columns":IN
FO$(4)=" keys"
   35 REM *** make selection ***
   40 MODE 7
```

```
   50 PRINT TAB(0,2)CHR$(136);CHR$(131);"S
elect search choice"
   60 PRINT TAB(2,4) CHR$(134);"1  Name of
 micro"
   70 PRINT TAB(2,6) CHR$(134);"2  Memory
size"
   80 PRINT TAB(2,8) CHR$(134);"3  No. of
columns"
   90 PRINT TAB(2,10) CHR$(134);"4  No. of
 keys"
  100 PRINT''
  110 CHOICE%=VAL(GET$)
  120 PRINT TAB(0,2)" "
  130 IF CHOICE%>4 OR CHOICE%<1 THEN RUN
  140 IF CHOICE%>1 THEN PROCLIMITS
  150 IF CHOICE%=1 THEN PROCNAME
  155 REM *** set up result screen ***
  160 MODE 3
  170 PROCHEADER
  180 PROCSTART
  190 REPEAT
  195   REM *** input data ***
  200   INPUT# DORIS,NAME$,MEM%,COL%,KEYS%
  205   REM *** check if data wanted ***
  210   IF CHOICE%=1 AND NAME$=N$ THEN PRO
CTELL
  220   IF CHOICE%=2 AND MEM%<ULIM% AND ME
M%>LLIM% THEN PROCTELL
  230   IF CHOICE%=3 AND COL%<ULIM% AND ME
M%>LLIM% THEN PROCTELL
  240   IF CHOICE%=4 AND KEYS%<ULIM% AND M
EM%>LLIM% THEN PROCTELL
  245   REM *** stop at end of file ***
  250   UNTIL EOF# DORIS
  260 CLOSE# DORIS
  265 REM *** any more? ***
  270 VDU26
  280 PRINT TAB(40)"Press 'M' for further
records"
  290 IF GET$="M" THEN GOTO 160
  300 CLS
  310 END

  320 DEFPROCLIMITS
  330 PRINT TAB(0,2+2*CHOICE%)CHR$(136)
  340 PRINT TAB(1,15) CHR$(131);"Give limi
ts:   Upper ";
  350 INPUT ULIM%
  360 PRINT TAB(16,16) CHR$(131);"Lower ";

  370 INPUT LLIM%
  380 ENDPROC
  390 DEFPROCNAME
  400 PRINT TAB(1,15) CHR$(131);"Give name
```

```
";
  410 INPUT N$
  420 ENDPROC
  430 DEFPROCSTART
  440 DORIS=OPENIN("Micros")
  450 ENDPROC
  455 REM *** print out wanted results ***
  460 DEFPROCTELL
  470 PRINT'NAME$,MEM%;INFO$(2),COL%;INFO$
(3),KEYS%;INFO$(4)
  480 ENDPROC
  490 DEFPROCHEADER
  500 VDU19,0,4;0;
  510 PRINT "Searching for ";
  520 IF CHOICE%=1 THEN PRINT N$:GOTO 540
  530 PRINT ULIM%;" >";INFO$(CHOICE%);" <
";LLIM%
  540 VDU 28,0,24,79,2
  550 ENDPROC
```

Even longer this time! Again, however, the program is relatively simple. There is not very much new here, just lots of it.

The first thing that the program does, after setting a few variables and dimensioning the odd array, is to request what it is going to be looking for. This program will search your database files for either a particular micro, if you give the name, or any micros that have a specification within a given range. First, then, the program asked you what you want to look for.

This is done in much the same way as we did in our program to search data statements. A menu is displayed and you choose options on the menu by their number, this is quite a friendly way of doing things, and is simple to implement. The choices are displayed with lines 40 to 100 and the selection is obtained with line 110. GET$ is used for inputting the option number because it mans that the user does not have to press <RETURN> after making their choice. Again a friendly point.

A few checks are then made on the number obtained. If it is out of range then the whole program is run again to get a new selection. If it is a choice of one of the specificiation details, then the desired range is input by PROCLIMITS. Now we are all set to look through the data.

A new screen is set up in PROCHEADER. Because a lot is going to be printed out on the screen, a large text format mode (3) is chosen. This also has the gaps between the lines of text. These are shown up by changing the background colour to blue as they help you to read across tables of figures. A heading is put at the top of the page to remind you what is being searched for. Then the text window is

changed to exclude the heading so that it stays on the screen as the results scroll away below it.

The real guts of the program is between lines 190 to 260. This is the part that actually reads in the data from the cassette and checks it for the required details. Line 200 reads in one record of data. One micro's worth. Then lines 210 to 240 check to see if either the name is the one wanted, if the first menu choice was made, or if the relevant specification is within the given range. If the data doesn't fit the bill then the loop just cycles around and the next record is looked at. If a suitable one is found, however, then PROCTELL is called. This prints out all the details about that micro onto the screen, along with suitable units.

This process is repeated until the end of the file is detected. This is done with a new Basic keyword, EOF#. EOF stands for 'end of file'. When the last bit of data has been read from a file, EOF# will become true. This has been used in this program simply to terminate the REPEAT-UNTIL loop by making EOF# the condition of the UNTIL part.

Of course you may have more than one file of micro data. In fact the chances are that you have added new files of data (using the previous program) on several occasions. Once one file is finished, then, you are asked by line 280 if there are any more to search through. If the answer is yes then the loop is repeated again.

When you run this program, with the data created by the previous one, you will find that the results come in starts. This is because of the queue system again. Just as the data is stored in a queue in the computer's memory for recording onto cassette, so the same thing happens when you read the data back in. A whole block is read in at once and put into a queue. The whole contents of this queue can then be processed by this program very quickly. There is then a wait while the next queue-full is loaded.

This database is certainly not the most powerful or flexible around. It will only cope with one sort of data — details on micros — and only four facts to each record. Real databases not only store the data itself but also how that data is to be used. This means that any subject can be catered for, in just about any format. However, such databases are very long programs indeed and are a little beyond the scope of this book!

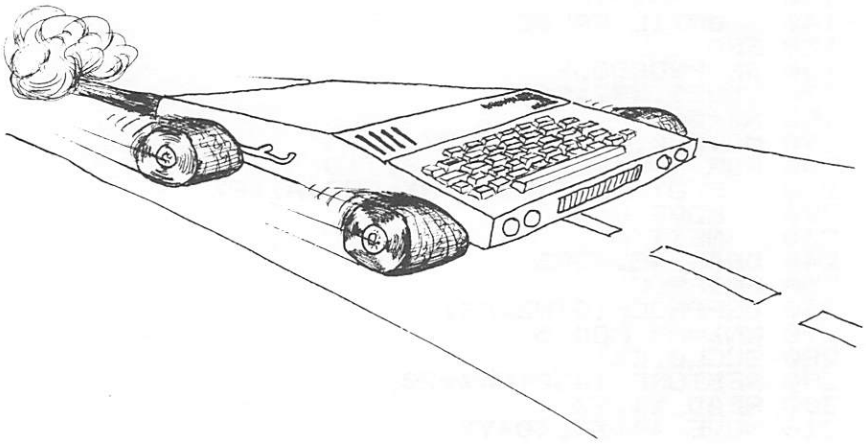
Our simple data storage system here should, however give you some idea as to what is possible with simple equipment and relatively simple programming. You will find that it is surprisingly easy to write your own data storage and sorting program to suit your own needs, whether that is a record collection or a sort of electronic diary. The chances are that no system based on a BBC micro and a cassette recorder will make life any easier than doing it

all by hand. However, it is certainly fun to do, and instructive as well.

In the next chapter we are going to look at something that the BBC micro is good at, whether equipped with a humble cassette recorder or super twin disk drives. All of the next chapter is about the screen. We are going to look at animated graphics on the BBC.

Chapter Nine

Animation and other games



As promised, we are now going to look at animation. There are several ways in which you can achieve very reasonable animation on the BBC micro. We are going to investigate a few of them. The first two methods are suitable for large displays of moving objects. The third and fourth method are more in the realm of games. We are going to use these methods to write a game later in this chapter. First though we'll animate a large object.

Everyone knows how basic film animation works. One frozen frame is shown, taken away, then the next is shown and so on. Animation on a micro uses much the same idea. A frame is drawn on the screen, then removed and then the next drawn. In this first method we will look at the simplest method of achieving this.

The easiest way to draw and erase different pictures in this way is do to just that. First draw a picture on the screen in a foreground colour, then erase it by drawing it again in the background colour, then draw the next frame in the foreground colour and so on.

This is what the first animation program does. Now we will create a program to produce an animated running stick man.

```
 10 MODE1
 20 N%=0
 30 FCOL%=3
 40 BCOL%=4
 50 VDU19,1,FCOL%;0;
 60 VDU19,3,FCOL%;0;
 70 VDU19,0,BCOL%;0;
 80 PROCBODY
 90 VDU29,640;512;
100 REPEAT
110   PROCFIG(N%,1)
120   PROCFIG(N%,0)
130   N%=N%+1
140   UNTIL FALSE
150 END
160 DEFPROCBODY
170 VDU29,592;900;
180 R%=50
190 MOVE 0,0:MOVE 0,R%
200 FOR A=0 TO 7 STEP PI/10
210   PLOT 85,SIN(A)*R%,COS(A)*R%
220   MOVE 0,0
230   NEXT A
240 DRAW 48,-388
250 ENDPROC
260 DEFPROCFIG(N%,C%)
270 NN%=N% MOD 5
280 GCOL0,C%
290 RESTORE (430+NN%*20)
300 READ X%,Y%
310 MOVE 40*X%,40*Y%
```

```
  320 FOR I%=1 TO 6
  330   READ X%,Y%
  340   DRAW 40*X%,40*Y%
  350   NEXT I%
  360 READ X%,Y%
  370 MOVE 40*X%,40*Y%
  380 FOR J%=1 TO 4
  390   READ X%,Y%
  400   DRAW 40*X%,40*Y%
  410   NEXT J%
  420 ENDPROC
  430 DATA 2,-9, 2,-8, -2,-5, 0,0, 0,-9, 0
,-9, -1,-10
  440 DATA -4,6, -1,4, -1,8, 0,4, -3,3
  450 DATA -2,-7, -1,-6, -4,-3, 0,0, 1,-5,
 3,-8, 2,-9
  460 DATA -6,7, -3,5, -1,8, 3,6, 1,4
  470 DATA -6,-6, -5,-6, -4,-2, 0,0, 2,-5,
 6,-6, 7,-7
  480 DATA -8,6, -4,5, -1,8, 3,7, 3,4
  490 DATA -5,-10, -5,-10, -4,-9, 0,0, 4,-
3, 8,-4, 9,-5
  500 DATA -6,6, -3,5, -1,8, 2,6, 0,3
  510 DATA -2,-10, -1,-10, -1,-5, 0,0, 1,-
5, 5,-8, 4,-9
  520 DATA -5,4, -2,4, -1,8, 1,5, -1,2
```

The main part of the program is in lines 10 to 140. The first 8 lines just set up the screen. Mode 1 is selected and then the background colour is redefined to blue and the foreground colours, 1 and 3 redefined to yellow. These are just chosen to look nice. If you want you can change these colours by changing the actual colours in lines 30 and 40.

The body of the man (a stick trunk and a solid circle head) is drawn with a call to the procedure, PROCBODY. These parts will not move and are drawn in logical colour 3. The circular head is drawn by the FOR-NEXT loop between lines 190 and 220. R% is the radius of the circle and the two co-ordinates, in the move of the graphics origin in line 160, are the co-ordinates of the circle's centre. If you understand a bit of trigonometry you should have little trouble following this one through, otherwise just accept it, and use it in your programs!

Once the screen has been set up and the body drawn, the animation of the arms and legs starts. This is done with repeated calls to a procedure called PROCFIG in lines 90 to 130.

The first call to PROCFIG draws one frame of the arms and legs movement in colour one. Then the next call to the procedure redraws the same frame in the background colour to erase the first picture.

The frame to be drawn is controlled with the number N%, passed

to the procedure (the other number passed is the colour used to draw the arms and legs — zero or one). Each time that the frame is drawn and erased the value of N% is increased by one. The next time around the REPEAT-UNTIL loop the next frame is drawn.

The frames are drawn with the procedure between lines 250 to 410 using the data at the end of the program. Although this procedure is quite long, its operation is very simple.

The first two lines of data represent the co-ordinates of the points (the joints) in the first frame (relative to a new graphics origin defined back in line 90). The next pair the next frame and so on. There are five frames in all. The FOR-NEXT loops between lines 310 and 400 read in the data, converts them to co-ordinates by scaling them with a factor of 40 and then drawing between the points.

Which data is used to draw the frame, in other words which frame is drawn, is controlled by the RESTORE command in line 280. This selects the data to be used in the same way as data statements were selected in the last chapter. Which data is selected depends on the value of N%. Although this starts at zero (line 20) and is increased throughout the program's RUN time the value of the line number to which the data pointer is RESTOREd (from NN%) cycles between zero and five, this is because the MOD function is used to obtain NN%. Remember MOD? We introduced this command back in chapter three.

That's how the animation works. RUN the program and you'll see it in action. You'll have to press <ESCAPE> to stop the program because the main program loop ends in UNTIL FALSE.

The trouble with this method of animation is that it is not very good! The resulting movie flickers a lot. This is because you micro takes a lot of time to draw out and erase each frame. Well, that time is still only a twinkling of an eye, but it is enough to spoil the effect. The next program cures this problem. The program is longer, although the aim of the program is still the same. You have to pay with program length for the better effect.

```
 10 MODE1
 20 N%=0
 30 FCOL%=3
 40 BCOL%=4
 50 VDU19,0,BCOL%;0;
 60 VDU19,3,FCOL%;0;
 70 VDU19,1,BCOL%;0;
 80 PROCBODY
 90 VDU29,640;512;
100 PROCFIG(N%,1)
110 REPEAT
```

```
  120    PROCFIG(N%+1,2)
  130    VDU19,1,BCOL%;0;
  140    VDU19,2,FCOL%;0;
  150    PROCFIG(N%,0)
  160    PROCFIG(N%+2,1)
  170    VDU19,1,FCOL%;0;
  180    VDU19,2,BCOL%;0;
  190    PROCFIG(N%+1,0)
  200    N%=N%+2
  210    UNTIL FALSE
  220  END
  230  DEFPROCBODY
  240  VDU29,592;900;
  250  R%=50
  260  MOVE 0,0:MOVE 0,R%
  270  FOR A=0 TO 7 STEP PI/10
  280    PLOT 85,SIN(A)*R%,COS(A)*R%
  290    MOVE 0,0
  300    NEXT A
  310  DRAW 48,-388
  320  ENDPROC
  330  DEFPROCFIG(N%,C%)
  340  NN%=N% MOD 5
  350  GCOL0,C%
  360  RESTORE (500+NN%*20)
  370  READ X%,Y%
  380  MOVE 40*X%,40*Y%
  390  FOR I%=1 TO 6
  400    READ X%,Y%
  410    DRAW 40*X%,40*Y%
  420    NEXT I%
  430  READ X%,Y%
  440  MOVE 40*X%,40*Y%
  450  FOR J%=1 TO 4
  460    READ X%,Y%
  470    DRAW 40*X%,40*Y%
  480    NEXT J%
  490  ENDPROC
  500  DATA 2,-9, 2,-8, -2,-5, 0,0, 0,-9, 0
,-9, -1,-10
  510  DATA -4,6, -1,4, -1,8, 0,4, -3,3
  520  DATA -2,-7, -1,-6, -4,-3, 0,0, 1,-5,
 3,-8, 2,-9
  530  DATA -6,7, -3,5, -1,8, 3,6, 1,4
  540  DATA -6,-6, -5,-6, -4,-2, 0,0, 2,-5,
 6,-6, 7,-7
  550  DATA -8,6, -4,5, -1,8, 3,7, 3,4
  560   DATA -5,-10, -5,-10, -4,-9, 0,0, 4,
-3, 8,-4, 9,-5
  570  DATA -6,6, -3,5, -1,8, 2,6, 0,3
  580  DATA -2,-10, -1,-10, -1,-5, 0,0, 1,-
5, 5,-8, 4,-9
  590  DATA -5,4, -2,4, -1,8, 1,5, -1,2
```

The program is almost the same. The difference lies in the main part of the program where the procedure to draw the arms and legs is called. Instead of just drawing then erasing each frame, the order now is a lot more complicated.

The figure is now drawn and undrawn out of sight. First a frame is drawn in colour one. Colour one is redefined to appear as the background colour so that you cannot see the comparatively slow drawing process. That frame is then flashed instantly onto the screen by again redefining it as the foreground colour (yellow). The next frame is then drawn in colour two, again invisibly. Colour one is then defined to be the background colour again and colour two the foreground. The effect of this is to swap over the frames instantaneously. The first disappears and the next appears. The first frame is then erased by redrawing it in logical colour zero (the background colour — blue). The next frame is then drawn in colour one, invisibly. The second and third frames are then swapped to make the third appear instantly. The second is erased and the fourth drawn in colour two. The whole process goes on like this for ever.

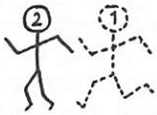
Now there's no disputing that that is a pretty complicated process. If you RUN the program, however, you can see the marked improvement that it has on the animation. As all the (visible) changes are now very swift the overall effect is much smoother.

Following this process is not very easy. Study the paragraph above and have a look at this diagram. Once you've mastered the overall idea, you'll find that doing your own animations in this way is very easy, and the results very reasonable.

Draw first frame invisibly

Reveal first frame

Draw second frame invisibly



Hide first frame. Reveal second frame

Undraw first frame

Draw third frame invisibly

Hide second frame. Reveal third frame

Undraw second frame etc...

The more complex the picture, of course, the longer the drawing and undrawing processes take, and the slower is the apparent motion. There's not a lot you can do about that. That's just one of the limitations of a micro. However you can rest with the assurance that, even if your latest attempt to animate a detailed rendition of the Mona Lisa comes out a bit on the lethargic side, BBC Basic is about the fastest around. Try it on any other home machine, and she'll be positively inanimate.

The trouble with the animation methods that we have covered so far is that they rely on the drawing and plotting commands of the BBC. Although these are very versatile, they are too slow for any complicated animation, like a game. For that we have to use character graphics. Because characters (either standard letters or your own user defined characters) are positioned on the screen as a single unit, this action is a lot faster.

There are two main methods of animation of characters. We are going to introduce both of these before this chapter is out. To do this we will write a game together. The game is going to use both of the forms of animation. It isn't anything fancy, you won't ever find this little masterpiece at your local arcade. The game is, however, reasonably short as a program. You can add your own extra features to it if you want to. We'll talk a bit about that later on. First though: the game.

We are going to write this game in sections. The structural nature of BBC Basic makes this quite easy to do. The final result will be a variation on the invading aliens theme. You will have a laser base to control that will move from side to side across the bottom of the screen, and hoards of marauding aliens will attack from the top.

Of course to play any action game such as this we need some way of controlling the goodies in their fight against the badies. This is usually done with a selection of keys on the keyboard taking the role of controls of your spacecraft or whatever.

We have already met some ways of using the keyboard for other things than typing. INPUT can be used to input the values of variables or the characters of string variables into a program. We could use this command for a game. Something like this

```
10 REPEAT
20    INPUT D$
30    IF D$="<" THEN PROCLEFT
40    IF D$=">" THEN PROCRIGHT
50    REM      other parts
60    REM        of the
70    REM          game
80    UNTIL FALSE
```

This is not a complete program so it won't RUN. It would work after a fashion if is was incorporated into a games program. PROCLEFT and PROCRIGHT would be procedures later in the program that would send your craft left and right respectively. The main problem with this method, though, is that it would require you to press <RETURN> after each operation of a control. This wouldn't give a very fast game. The program would also wait for your control signal each time the INPUT command was reached. This means that you would have to continually command turn left or right for the rest of the game program ever to be run.

We have also met the command GET. We could similarly write a game to include this command like this

```
10 REPEAT
20    G=GET
30    IF G=60 THEN PROCLEFT
40    IF G=62 THEN PROCRIGHT
50    REM      other parts
60    REM        of the
70    REM          game
80    UNTIL FALSE
```

The ASCII codes for the <key and the> key are 60 and 62. This would solve the problem of having to press <RETURN> but the program would still wait at line 20 for your command. You couldn't just sit back and watch it all happen before you. What we need is a command that will just look at the keyboard to see what is going on.

There is, of course, just such a command. This is INKEY. INKEY is similar to GET in that it comes in two forms. Numeric and string. INKEY and INKEY$. We use it in a similar way by assigning a variable with the value INKEY. Following the command there must be a number inside a pair of brackets. Like this

```
I%=INKEY(30)
I$=INKEY$(100)
```

This number represents a time limit. The function INKEY will cause the micro to wait for a period of time for a key to be pressed. Then it will return the ASCII code or the character of the pressed key. If no key is pressed then a value of −1 is returned for INKEY and an empty string for INKEY$.

The period of time that the micro will wait is given by the number inside the brackets. This is measured in centi-seconds, just like TIME itself.

Your BBC micro has a bit of its memory set aside to temporarily store whatever you type in at the keyboard. This is called the 'keyboard buffer'. This means that you can type ahead of the

computer. All it has to do is to look at this bit of memory to see what characters it has missed you typing while it was busy. This is very convenient for fast typists, but it does mean that the INKEY function will return the keys that you may have typed some time ago without waiting for the one that you really mean, now. When we use the INKEY or INKEY$ function we should clear out the keyboard buffer first.

This action is rather like the way the sound queue (another buffer) is flushed to make a sound appear instantly.

The command to do this is *FX 15,1. We could also use *FX 15,0. This clears all the buffers used by the micro including, for example, the sound buffer. This has the unfortunate effect of stopping any sound that is being produced whenever we want to flush the keyboard buffer. So we'll stick with *FX 15,1. This just flushes whichever buffer that we are concerned with at the time.

The format of this command is a little strange. Starting off with a * is odd, isn't it. There are in fact many *FX commands. The FX is short for 'effects'. these are not really Basic comands at all but commands that are passed directly to the operating system. They can be used from a Basic program, though so we will treat them the same. We will be meeting more of these as we go on, and you will find a complete list in the appendix.

Okay, so now we can use INKEY for our game program. INKEY requires no <RETURN> and the program does not have to wait for our response because we can quite happily use INKEY(0) so that only keys that we are pressing at the instant that the program line is processed will be noticed.

We still have a problem, though. When you're writing a program and you keep your finger on a key a single character is produced on the screen and then after about half a second's wait a whole string of characters are produced at the rate of about twenty-five a second. This is the auto-repeat facility of the keyboard.

Now that mass of characters will be flushed from the keyboard buffer, but what if the program happens to look to the keyboard for its instantaneous INKEY(0) input when the gap before the rush is being played out? Your carefully considered control of your spacecraft will go unnoticed.

Naturally there is a way around this trouble too. There is another FX command, *FX 11 that sets the delay before the flood of characters appears. We can't reduce the delay to nothing because *FX 11,0 just switches off the delay altogether. This would mean you would have to repeatedly press the < key to turn left through a large angle. What we want to do then is to reduce the delay before the auto-repeat to a very small value. Just for good measure we can also increase the repetition rate of the flood of characters. Then we

would be sure of every touch on the controls, long or short, being acted on properly. We can change the time between each repetition with *FX 12. Our change to the keyboard would therefore be something like

```
*FX 11,1
*FX 12,1
```

The time periods are both measured in centi-seconds. Just try setting the keyboard repeat in this way to see the results. To restore the auto-repeat to its default setting type in *FX 12,0. You might find this a bit hard with the keyboard in its present state so perhaps you had better just press <BREAK>.

All this probably seems to you as rather a complicated method to make a program see what keys you are pressing at any time. It also has the decided disadvantage of not being able to detect two keys down at the same time. It will only notice the last one pressed. Because of this, we don't usually use this method at all. Instead we use yet another variation of INKEY. This time with a negative number inside the brackets.

Used in this way, INKEY acts entirely differently. Now this function just returns a yes or no answer. For every key on the keyboard there is a negative number to put in an INKEY statement like this

```
IF INKEY(-103) THEN PROCLEFT
```

103 is the number for the < key. It doesn't matter whether the key is pressed in conjunction with the shift or not. It is the key itself that is being detected and not the character. In fact even the shift keys can be detected in this way if their number (−1) is put in the brackets or the <RETURN> (−74) or even the <CAPS LOCK> (−65). You will find all the numbers on page 275 of your User Guide.

In the program line, above, the INKEY(−) is used, as it normally is, in an IF–THEN statement. If the < key is down, when this line is reached, then the program will jump to PROCLEFT otherwise it will carry on regardless.

Two or more keys can be detected at once

```
IF INKEY(-103) AND INKEY(-1) THEN PROCUPLE
FT
IF INKEY(-103) AND NOT INKEY(-1) THEN PROC
DOWNLEFT
```

In fact this method of key detection is about the most versatile to be found on any home computer. That is why we are going to use it for our game.

The first thing to do in our game program is to set up a

background of some kind. We will use a plain blue screen as our space backdrop. Next we must give you your laser base. For the base we'll use a single character. The shape of the character we will define using the VDU 23 command.

We shall be using the simplest form of animation for the laser base. To move the base character the program will simply print a space in the old position and the base character in the new, moved position.

To get a continual control of its movements we simply enclose this process in a REPEAT–UNTIL loop with a negative INKEY key control to decide the direction of the move. We have used the Z and X keys to move the laser base left and right.

```
  100 MODE2
  200 PROCSETUP
  300 REPEAT
  400   PROCERASE
  500   PROCKEYS
  600   PROCPUT
  700   T%=TIME+2:REPEAT UNTIL TIME>T%
  800   UNTIL FALSE
  900 END
 1000 DEFPROCSETUP
 1100 COLOUR 132:CLS
 1200 COLOUR 3
 1300 VDU23,224,24,24,24,126,219,165,219,1
26
 1400 X%=10:Y%=30
 1500 ENDPROC
 1600 DEFPROCLEFT
 1700 X%=X%-1
 1800 IF X%<0 THEN X%=0
 1900 ENDPROC
 2000 DEFPROCRIGHT
 2100 X%=X%+1
 2200 IF X%>19 THEN X%=19
 2300 ENDPROC
 2400 DEFPROCERASE
 2500 PRINT TAB(X%,Y%);CHR$(32);
 2600 ENDPROC
 2700 DEFPROCPUT
 2800 PRINT TAB(X%,Y%);CHR$(224);
 2900 ENDPROC
 3000 DEFPROCKEYS
 3100 IF INKEY(-98) THEN PROCLEFT
 3200 IF INKEY(-67) THEN PROCRIGHT
 3300 ENDPROC
```

This program has been written with line numbers one hundred apart so that you can easily see the changes (numbered as units) that are made as we go in.

The screen is coloured and the base shape defined in PROCSETUP. The position of the base at any one time is held in two variables, X% and Y%. The base is moved by removing it (PROCERASE), changing the values of X% and Y% if the keys are pressed (PROCKEYS) and then printing the base again at the new position (PROCPUT). The printing of the base at the right position is done with a TAB statement in PROCERASE and PROCPUT.

Although the keyboard is only actually checked for your control during the operation of PROCKEYS, the whole REPEAT–UNTIL loop cycles around so fast that it gives the impression that you have control the entire time.

PROCKEYS uses the negative INKEY to determine which, if either, of the two procedures, PROCLEFT or PROCRIGHT to call. They are called from within the procedure, PROCKEYS. This is nesting that we met earlier.

Both PROCLEFT and PROCRIGHT work in much the same way. One is added or taken away from the value of X%. The result is then checked to see if this would take the base off either edge of the screen. If it would then the value of X% is corrected to keep the base teetering on the edge. This checking can, in fact, be combined with the change to X%. Both lines, 1700 and 1800, for example, could be replaced with one single line.

```
1800 X%=X%-1-(X%=0)
```

This looks a little strange at first, but it is basically very simple. Your BBC micro has an idea of what is true or false. However, unlike us humans, the micro assigns a value to the concept of true (−1) and a value to the concept of false (0). Ask your micro to PRINT TRUE and PRINT FALSE to see this.

What our replacement line is doing then is this. If the current value of X% is more than zero, say five, then the statement inside the brackets is not true. 5=0 is false. The whole line then reads X%=5−1−(0). X% therefore now takes the value 4 and so the base is moved a bit to the left as you would expect. If however the base is at the left hand edge of the screen, X% now has a value of zero and so the bracket becomes true and has a value of minus one. Now the micro reads the line as X%=0−1−(−1), which is zero. So the base stays where it is.

The same thing can be done for lines 2100 and 2200. They could be replaced with this line.

```
2200 X%=X%+1+(X%=19)
```

This use of TRUE and FALSE can make life much easier when programming and the programs very much faster. However we are

going to leave the edge checking as it is so that the program is easier to read. We don't really need any more speed in this program.

That's the game so far. If you RUN it you'll see that there are still a few problems (you'll have to press <ESCAPE> to stop). For a start the cursor follows the base around back and forth across the screen. This doesn't look very nice at all.

The base also seems to flicker a lot. This is because all the program is done in between the time that the base is erased and the time that the base is printed again in its new position.

We can solve this by introducing the idea of the last position of the base. Each time around the loop we will transfer the values of X% to another variable, OX%. Now we don't erase the base straight away but wait until we have worked out the new position.

The erasing and printing therefore follow one another closely so the flicker is greatly reduced. Try it. Change your program to look like this:

```
  100 MODE2
  200 PROCSETUP
  300 REPEAT
  400   OX%=X%
  500   PROCKEYS
  505   PROCERASE
  600   PROCPUT
  700   T%=TIME+2:REPEAT UNTIL TIME>T%
  800   UNTIL FALSE
  900 END
 1000 DEFPROCSETUP
 1100 COLOUR 132:CLS
 1200 COLOUR 3
 1205 VDU23;11,0;0;0;0;
 1300 VDU23,224,24,24,24,126,219,165,219,1
26
 1400 X%=10:Y%=30
 1405 ENVELOPE 1,1,0,0,0,1,1,1,127,-1,-1,-
1,126,110
 1406 ENVELOPE 3,1,2,2,-2,2,2,4,126,0,0,0,
80,80
 1407 SOUND 3,3,50,255
 1500 ENDPROC
 1600 DEFPROCLEFT
 1700 X%=X%-1
 1800 IF X%<0 THEN X%=0
 1900 ENDPROC
 2000 DEFPROCRIGHT
 2100 X%=X%+1
 2200 IF X%>19 THEN X%=19
 2300 ENDPROC
 2400 DEFPROCERASE
 2505 PRINT TAB(OX%,Y%);CHR$(32);
 2600 ENDPROC
```

```
2700 DEFPROCPUT
2800 PRINT TAB(X%,Y%);CHR$(224);
2900 ENDPROC
3000 DEFPROCKEYS
3100 IF INKEY(-98) THEN PROCLEFT
3200 IF INKEY(-67) THEN PROCRIGHT
3205 IF INKEY(-1) THEN PROCFIRE
3300 ENDPROC
3400 DEFPROCFIRE
3500 SOUND &11,1,10,40
3600 SOUND &10,1,6,40
3700 ENDPROC
```

We have had to change PROCERASE to erase at the old position (given by OX%) and change the order of the main part of the program. The command to remove the cursor is in line 150. This isn't a normal VDU 23 command but is the only satisfactory method of removing the text cursor. It is a pity that a more friendly command couldn't have been provided, but still. You'll just have to learn this one. We have also added a couple more features.

We have given the game a basic background sound. This is done in lines 1406 and 1407. The SOUND statement uses a duration of 255 which means that the noise continues for ever, or in this case the duration of the game.

We have also added the start of your fire power. In PROCKEYS there is now a line (3205) to detect the shift key being pressed. If this is pressed, the procedure, PROCFIRE is called. As yet this procedure only produces a shot noise. We have to extend it a bit to actually produce a missile.

We will design the game to only allow one bullet in the air at a time. This not only makes the game a bit harder to play but also much easier to program. The first thing we will add to PROCFIRE then is a check to see if another bullet is going at the time. We'll use a variable as a 'flag' to tell us if a bullet is in the air. We'll call this flag FIRE%. When the PROCFIRE is first called this flag will be set to TRUE. At the start of the procedure we'll include a test on that flag. If it is set the procedure will end there.

We also need to put a bullet in the air and move the bullet. For this we have another procedure, PROCBULLET. The shape of the bullet character (225) is defined in PROCSETUP. PROCBULLET is called each time around the REPEAT–UNTIL loop if the flag, FIRE% is set.

This procedure moves the bullet character one position up the screen using the same animation method as we used for the base. There is a test in the procedure for the bullet reaching the top of the screen. If it has, then another (yet another!) procedure called BULLETEND is called which erases the last bullet character and

resets the flag FIRE% to false ready for the next bullet.

These are the program lines that will add these features. Try it out.

```
1305 VDU23,225,0,24,24,24,24,60,36,0
1404 FIRE%=FALSE
3206 IF FIRE% THEN PROCBULLET
3405 IF FIRE% THEN ENDPROC
3605 FIRE%=TRUE
3606 BX%=X%:BY%=Y%
3800 DEFPROCBULLET
3900 PRINT TAB(BX%,BY%);CHR$(32)
4000 BY%=BY%-1
4100 IF BY%=0 THEN PROCBULLETEND:ENDPROC
4200 PRINT TAB(BX%,BY%);CHR$(225)
4300 ENDPROC
4400 DEFPROCBULLETEND
4500 FIRE%=FALSE
4600 ENDPROC
```

So far, so good. However with all this fire power we could do with an enemy! We have to add another line to PROCSETUP to define the invader character and then a procedure to print the alien line-up at the start of the game. This is called PROCALIENS. All it does is use a FOR–NEXT loop to place an alien at each of ten character positions across the top of the screen. This is done using the graphics cursor (after VDU 5) and GCOL 3,5. The GCOL 3 is used because we are going to use a different kind of animation.

Remember how we found that GCOL 3 will remove an object from the screen if the object is printed twice? That's how we are going to animate the aliens.

To move an alien we just have to print the character once in the same place to remove the old one and once in the new position to place the 'new' alien on the screen. Although colour five is used in the GCOL statement, the aliens come out red. This is one of the effects of GCOL 3.

To move the aliens we use another procedure, PROCALIENMOVE. Each time around the main REPEAT–UNTIL loop this procedure is called. Each time that it is called one alien is moved randomly downwards. The alien that is moved is also chosen randomly from the eleven on the screen. All the co-ordinates of the aliens are kept in two arrays called AX% and AY%. These arrays have to be dimensioned with length of eleven each, at the beginning of the program.

This means that once the alien to be moved has been chosen, the coordinates can be simply looked up from the arrays and the new co-ordinates, once it has been moved, stored back there. Add these latest additions to your program.

```
 1306 VDU23,226,60,90,90,126,126,36,66,129
 1408 DIM AX%(10),AY%(10)
 1409 PROCALIENS
 3207 PROCALIENMOVE
 4700 DEFPROCALIENS
 4800 VDU5
 4900 GCOL 3,5
 5000 FOR I%=0 TO 10
 5100   AX%=128+I%*128
 5200   MOVE AX%,928:PRINT CHR$(226)
 5300   AX%(I%)=AX%:AY%(I%)=928
 5400   NEXT I%
 5500 VDU4
 5600 ENDPROC
 5700 DEFPROCALIENMOVE
 5800 N%=RND(11)-1
 5900 XM%=64*(RND(3)-2)
 6000 VDU5
 6100 MOVE AX%(N%),AY%(N%)
 6200 PRINT CHR$(226)
 6300 IF AX%(N%)<64 OR AX%(N%)>1216 THEN G
OTO 6600
 6400 AX%(N%)=AX%(N%)+XM%
 6500 AY%(N%)=AY%(N%)-64
 6600 MOVE AX%(N%),AY%(N%)
 6700 PRINT CHR$(226)
 6800 VDU4
 6900 ENDPROC
```

RUN this program and you see the aliens move down towards your base. You can fire at them but it has little effect as we have not yet added any part of the program to detect a bullet hitting an alien invader.

We must add to the procedure that moves the bullet (PROCBULLET) a check to see if it is about to hit an alien. To do this we use the POINT function. This returns the colour of a point on the screen. The point that we want is defined by its x and y co-ordinates. The point's co-ordinates are given as graphic co-ordinates for greatest accuracy.

For this program we want a point just above the bullet's current position. This is found by multiplying the horizontal and vertical position of the bullet by the width of one character position (64 and 32 respectively in mode 2). Remember to make adjustment for the fact that the graphics y co-ordinates start at the bottom but the character vertical positions start at the top of the screen.

If the colour red is detected in front of the bullet then PROCHIT is called. This procedure has to do three things. It has to reset the flag FIRE% so that a further bullet can be fired. It has to find out which of the aliens has been hit and remove that one from the screen, and it also has to make some sort of noise suitable for a dying alien.

The procedure, PROCHIT, works out which alien is hit by going through all of the alien x and y co-ordinates to see which ones correspond to the point above the bullet's position.

Now we have to have some method of telling the program that a particular alien is dead or alive so that the procedure, ALIENMOVE, only moves and displays the aliens that should still be on the screen. For this we will use more flags. One for each alien. These flags are kept as an array (D%) and they must all be set to FALSE at the beginning of the program so that all the aliens can be displayed initially.

The array must also be dimensioned at the beginning of the program in the DIM statement.

In the procedure, PROCALIENMOVE, we include a check on the randomly chosen alien's flag. If this is TRUE (if the alien has been hit) we end the procedure there.

The sound of a dying alien that this program uses is perhaps a little lacking in realism. It is a short high pitched beep. If you want to add your own sound then you can do that quite easily.

PROCHIT has one other function too. It increases a variable called SC%. This is your score. SC% must be set to zero at the beginning of the game so a line to do this is included in PROCSETUP.

As well as detecting the aliens' demise we have to do something about their drifting off the bottom of the screen. For this we include a simple check in PROCALIENMOVE. After the alien in the old position has been erased with the first print in the procedure, a check is made to see if the new position will take the alien below the level of your base. If it does, then the flag for that alien is set to FALSE so that it won't trouble your screen again and then the procedure ends.

Sounds really simple doesn't it! Actually it isn't all that complicated. You can see that if you make these changes to your program.

```
 1403 SC%=0
 1408 DIM AX%(10),AY%(10),D%(10)
 4105 IF POINT(64*BX%+32,32*(33-BY%))=1 TH
EN PROCHIT:ENDPROC
 5300 AX%(I%)=AX%:AY%(I%)=928:D%(I%)=FALSE
 5805 IF D%(N%) THEN ENDPROC
 6505 IF AY%(N%)<128 THEN D%(N%)=TRUE:GOTO
 6800
 7000 DEFPROCHIT
 7100 FIRE%=FALSE
 7200 FOR I%=0 TO 10
 7300    IF AX%(I%)<>64*BX% OR AY%(I%)<>32*
(33-BY%) THEN GOTO 8000
 7400    D%(I%)=TRUE
 7500    VDU 5
```

```
 7600    MOVE AX%(N%),AY%(N%):PRINT CHR$(22
6)
 7700    VDU 4
 7800    SOUND &11,-15,200,5
 7900    SC%=SC%+50
 8000    NEXT I%
 8100 ENDPROC
```

Your complete program, renumbered, should look like this

```
   10 MODE2
   20 PROCSETUP
   30 REPEAT
   40    OX%=X%:OY%=Y%
   50    PROCKEYS
   60    PROCERASE
   70    PROCPUT
   80    T%=TIME+2:REPEAT UNTIL TIME>T%
   90    UNTIL FALSE
  100 END
  110 DEFPROCSETUP
  120 COLOUR 132:CLS
  130 COLOUR 3
  140 VDU23;11,0;0;0;0;
  150 VDU23,224,24,24,24,126,219,165,219,1
26
  160 VDU23,225,0,24,24,24,24,60,36,0
  170 VDU23,226,60,90,90,126,126,36,66,129
  180 X%=10:Y%=30
  190 SC%=0
  200 FIRE%=FALSE
  210 ENVELOPE1,1,0,0,0,1,1,1,127,-1,-1,-1
,126,110
  220 ENVELOPE3,1,2,2,-2,2,2,4,126,0,0,0,8
0,80
  230 SOUND3,3,50,255
  240 DIM AX%(10),AY%(10),D%(10)
  250 PROCALIENS
  260 ENDPROC
  270 DEFPROCLEFT
  280 X%=X%-1
  290 IF X%<0 THEN X%=0
  300 ENDPROC
  310 DEFPROCRIGHT
  320 X%=X%+1
  330 IF X%>19 THEN X%=19
  340 ENDPROC
  350 DEFPROCERASE
  360 PRINT TAB(OX%,OY%);CHR$(32);
  370 ENDPROC
  380 DEFPROCPUT
  390 PRINT TAB(X%,Y%);CHR$(224);
  400 ENDPROC
  410 DEFPROCKEYS
```

```
  420 IF INKEY(-98) THEN PROCLEFT
  430 IF INKEY(-67) THEN PROCRIGHT
  440 IF INKEY(-1) THEN PROCFIRE
  450 IF FIRE% THEN PROCBULLET
  460 PROCALIENMOVE
  470 ENDPROC
  480 DEFPROCFIRE
  490 IF FIRE% THEN ENDPROC
  500 SOUND &11,1,10,40
  510 SOUND &10,1,6,40
  520 FIRE%=TRUE
  530 BX%=X%:BY%=Y%-1
  540 ENDPROC
  550 DEFPROCBULLET
  560 PRINT TAB(BX%,BY%);CHR$(32)
  570 BY%=BY%-1
  580 IF BY%=0 THEN PROCBULLETEND:ENDPROC
  590 IF POINT(64*BX%+32,32*(33-BY%))=1 TH
EN PROCHIT:ENDPROC
  600 PRINT TAB(BX%,BY%);CHR$(225)
  610 ENDPROC
  620 DEFPROCBULLETEND
  630 FIRE%=FALSE
  640 ENDPROC
  650 DEFPROCALIENS
  660 VDU5
  670 GCOL 3,5
  680 FOR I%=0 TO 10
  690   AX%=128+I%*128
  700   MOVE AX%,928:PRINT CHR$(226)
  710   AX%(I%)=AX%:AY%(I%)=928:D%(I%)=FAL
SE
  720   NEXT I%
  730 VDU4
  740 ENDPROC
  750 DEFPROCALIENMOVE
  760 N%=RND(11)-1
  770 IF D%(N%) THEN ENDPROC
  780 XM%=64*(RND(3)-2)
  790 VDU5
  800 MOVE AX%(N%),AY%(N%)
  810 PRINT CHR$(226)
  820 IF AX%(N%)<64 OR AX%(N%)>1216 THEN G
OTO840 860
  830 AX%(N%)=AX%(N%)+XM%
  840 AY%(N%)=AY%(N%)-64
  850 IF AY%(N%)<128 THEN D%(N%)=TRUE:GOTO
 880
  860 MOVE AX%(N%),AY%(N%)
  870 PRINT CHR$(226)
  880 VDU4
  890 ENDPROC
  900 DEFPROCHIT
  910 FIRE%=FALSE
```

```
  920 FOR I%=0 TO 10
  930    IF AX%(I%)<>64*BX% OR AY%(I%)<>32*
(33-BY%) THEN GOTO 1000
  940    D%(I%)=TRUE
  950    VDU5
  960    MOVE 64*BX%,32*(33-BY%):PRINTCHR$(
226)
  970    VDU4
  980    SOUND &11,-15,200,5
  990    SC%=SC%+50
 1000    NEXT I%
 1010 ENDPROC
```

Well that's about it really. All we need now is something to round the game off. At the moment, when all the aliens have been shot or have drifted off the bottom of the screen, the game just continues with nothing for you to shoot at.

We have to detect when all of the aliens have vanished so that we can end the game. To do this we just keep a count of the aliens as they vanish. A variable, D% is set to the initial number of aliens right at the start, in PROCSETUP. Each time that an alien is shot or moves off the screen, one is taken from this variable. The main REPEAT–UNTIL loop is now changed to cycle until D% is equal to zero. Until the aliens are all gone.

After the loop we put a call to a final procedure to clear the screen, print your score, and ask you for another game. This procedure we call, reasonably enough, PROCEND. Here are the lines to add.

```
   90 UNTIL DEAD%=0
   95 PROCEND
  255 DEAD%=11
  890 IF AY%(N%)<128 THEN D%(N%)=TRUE:DEAD
%=DEAD%-1:GOTO 890
  985 DEAD%=DEAD%-1
 1030 DEFPROCEND
 1040 CLS
 1050 SOUND &13,0,0,0
 1060 PRINT TAB(4,10);"Score - ";SC%
 1070 T%=TIME+200:REPEAT UNTIL TIME>T%
 1080 PRINT TAB(4,15);"another go ?";
 1090 *FX 15,0
 1100 IF GET$="Y" OR GET$="y" THEN RUN
 1110 ENDPROC
```

There is, of course a lot more that we could add to this program. You could add a running score on the screen, a bit of a delay between games, even aliens that shoot back! However, hopefully the game as it is gives you an idea both of animation on the BBC and just how much you can do with BBC Basic.

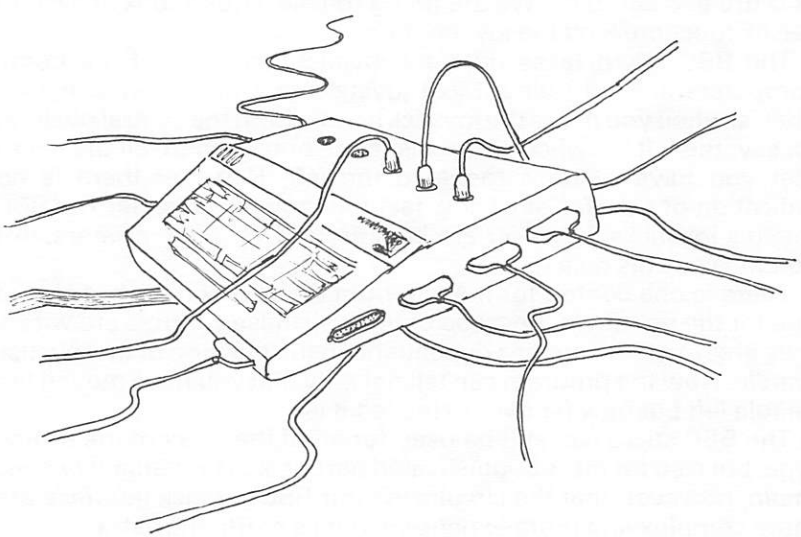
There is perhaps one addition to the program that we should

mention. So far we have controlled the movements and fire power of your base with three keys on the keyboard. There is however another method. Joysticks.

Joysticks are among the many different accessories you can connect on to your BBC micro. In the next chapter we are going to have look at some of them. Meanwhile, have fun with our game.

Chapter Ten

Out and about



In this, final, chapter we are going to investigate ways in which your BBC micro can draw information from and affect the outside world. So far we have concerned ourselves exclusively with internal matters. All our programs have gone only to the keyboard or cassette recorder for input, and all output has been to the screen or loundspeaker. Now is the time to change all that.

All our programs to date have been decidedly on the trivial side, and all the more enjoyable for that. However all the so-called serious uses of microcomputers tend to require the outside world to play a bigger part in programs than we have allowed for so far. The BBC micro is unusually well equipped for this in the field of home computers.

Your BBC micro comes complete with a wide range of interfaces. These allow you to connect your micro to many other pieces of equipment, printers, monitors, joysticks, sensors, and many specialised items.

The interfaces that are provided in your micro are simply electronic circuits to adapt the signals that the microprocessor inside your micro deals with, to those required by other pieces of machinery. The interfaces on the BBC micro are to be found along the back panel and underneath the cabinet, behind the keyboard.

The sockets along the back panel are, from left to right, UHF out, video, RGB, RS423, Cassette, analogue in, and econet. The last is a only a labelled punch-out hole. Underneath there are five sockets labelled disc, printer, user input/output, 1MHz bus, and tube.

Some of these interfaces are simple to use and some are more obscure and complex. We are going to take a quick look at each to see its function. First the joystick port.

The BBC micro takes different joysticks to most of the home computers around today. Most joysticks are based on switches. That is, when you move the joystick handle from the central position to, say, the left, a switch contact is made or broken to tell the micro that you have made a move to the left. However there is no indication of how far or of how fast the move was made. The BBC micro's joysticks, however are built around variable resistors, like volume controls on a hifi.

There is one control for the left–right direction on each stick and one for the up–down. Because of the way these controls are wired, they give out two voltages dependent on the position of the joystick handle. Now the program can tell not only that you have moved the handle left but how far over to the left it is.

The BBC sticks can still be used for all of the tasks of the switch type, but also for more sophisticated purposes. This versatility does mean, however, that the circuits in your BBC joystick interface are more complex and more expensive, but its worth the extra.

There are four voltage measuring circuits in your micro to detect the voltages from the two controls on each of the pair of joysticks. Unfortunately there is no real standard for which of these 'channels' should control which direction on which stick. This means that for any one program, joysticks from different manufacturers have to be held in different orientations to function correctly. However we are going to assume a standard. If your joysticks do not work correctly with the programs that follow, you will have to change the programs slightly.

In fact the User Guide specifies a different standard but the joystick manufacturers do not seem to have noticed this! Most sticks work to our standard.

The voltages are detected with what is known as an 'analogue to digital converter' (ADC). This is a wonderful little chip that takes in analogue signals and outputs the equivalent digital signal that your computer can understand. You need not really understand the difference between analogue and digital signals except that you should realise that almost all measurements that we can take in the real world are analogue and these can only be understood by a micro if they are suitably converted to digital signals.

The digital signals that come out of your micro's ADC are numbers in the range 0 to 65520. These can be any multiple of 16. So the number that results from a low voltage is 0, from the highest that it can detect, 65520, and from a middle voltage, 32752. These numbers are not very convenient but you are stuck with them. That is the way the ADC is constructed.

Our standard will be like this

	joystick 1	joystick 2
left	ch 1 high	ch 3 high
right	ch 1 low	ch 3 low
up	ch 2 high	ch 4 high
down	ch 2 low	ch 4 low

All the intermediate positions give intermediate values on the relevant channels.

Okay, so how do we actually read the values that the joystick gives us? The answer, of course is that there is another Basic keyword to do this. This is ADVAL, which stands for Analogue to Digital converter VALue. We use this keyword to assign the value to a variable. We specify which channel we are talking about by putting the channel nbumber (1 to 4) in brackets after the ADVAL. This short program illustrates the use of ADVAL.

```
10 MODE 4
20 REPEAT
30   PRINT ADVAL(1),ADVAL(2),ADVAL(3),A
DVAL(4)
40   T%=TIME+10:REPEAT UNTIL TIME>T%
50   UNTIL FALSE
60 END
```

The program continually prints out the values from all four channels. While the program is running, play around with your joystick or joysticks. You should find that, for example, when you move the number one stick to the left the first column turns to a succession of very high numbers, around 65520. If you push it to the left, this column turns to low numbers, around 80. It is unlikely that you will ever get exactly zero with either stick. This does not mean that the stick or the micro is faulty. The 65520 value represents a voltage of only 1.8 volts. Each step of 16 in the value, then, corresponds to a voltage step of only 0.0004 volts. It therefore only needs the tiniest of voltage leaks to push the ADVAL value up from zero.

The primary use for joysticks is the control of games. You can see how continual calls to the joystick interface instead of the keyboard will be able to return numbers to the program to control the movements of a games character. However for games we need to be able to fire missiles, laser bolts, and so forth as well as move. On each of your joysticks there is a fire button. This is simply a switch, so the signals from it don't need to be converted to digital signals. For convenience, however, these signals are accessed from a program using the ADVAL command. We use channel zero to both sticks' fire buttons.

The way in which we use it is a bit complicated. However, here goes. We need to use the function, AND, on the channel zero result. We might use a program line like this

```
100 SHOOT%=ADVAL(0) AND 3
```

After this, the value of the variable SHOOT% will tell us which of the buttons, if either, are being pressed at that instant.

SHOOT% = 0	Neither pressed
SHOOT% = 1	No. 1 stick fire button pressed
SHOOT% = 2	No. 2 stick fire button pressed
SHOOT% = 3	Both fire buttons pressed

In the last chapter we wrote a game to demonstrate animation. We used the keyboard to control the position of our base and to fire

at the advancing aliens. Now we can convert that program to work with a joystick. Add these lines to your old program.

```
420 IF ADVAL(1)>43680 THEN PROCLEFT
430 IF ADVAL(1)>21840 THEN PROCRIGHT
440 IF (ADVAL(0) AND 3)=1 THEN PROCFIRE
```

We are using the joysticks in the same way as the switch type. You must move the stick some way over to the left or right to move the base. The figures, 43680 and 21840 were chosen because they are ⅓ and ⅔ the range of possible values. You can alter the 'feel' of the joystick control by altering these figures.

We could change the program so that the base's position is directly linked to the position of the joystick. We would take the bases co-ordinates from the ADVAL value.

```
420 J%=ADVAL(1)
430 X%=(65520-J%)*19/65520
```

Now the base's co-ordinate (X%) is obtained directly from the ADVAL value, J%, with suitable scaling to convert it from a 0–65520 number to a 0–19 number ready for use in the TAB. PROCLEFT and PROCRIGHT are now no longer needed.

On the whole, this last method of control is not very suitable for games. Games require a large shift of the joystick to send the base, or whatever, a small distance, like our first method. That is why most home computers use switch type joysticks that do this job very well. However if the program is not a game, but some task that requires accurate positioning of the cursor on the screen, then a method that makes full use of the BBC micro joysticks can be useful. We can demonstrate this with a program that turns your screen into a drawing board.

```
  10 MODE1
  20 PROCSETUP
  30 REPEAT
  35   REM *** erase old cross ***
  40   PROCPRINT
  45   REM *** scale joystick readings **
*
  50   X%=(65520-ADVAL(1))*1279/65520
  60   Y%=ADVAL(2)*1023/65520
  65   REM *** print new cross ***
  70   PROCPRINT
  80   UNTIL FALSE
  90 END
 100 DEFPROCSETUP
 110 VDU 23,224,0,24,24,102,102,24,24,0
 120 VDU 5
 130 GCOL 3,3
```

```
135 REM *** draw border ***
140 DRAW 0,1023:DRAW 1279,1023
150 DRAW 1279,0:DRAW 0,0
155 REM *** initiallise cross ***
160 X%=640:Y%=512
170 PROCPRINT
180 ENDPROC
190 DEFPROCPRINT
200 MOVE X%-16,Y%+16
210 PRINT CHR$(224)
220 ENDPROC
```

We are using the GCOL 3 method of animation like we used for the aliens in our game in the last chapter.

PROCSETUP defines a character the shape of a cross as your sight and then draws a border around the screen. The main part of the program loops around lines 40 to 70.

Each time around the loop the cross is erased, its co-ordinates are changed and then the cross is reprinted in the new position. The erasure and printing is done with the procedure, PROCPRINT. The co-ordinates are changed by lines 50 and 60. The position of the joystick is read and the ADVAL number scaled appropriately to fit within the co-ordinate range of the screen.

So far this program just moves a cross-shaped cursor around the screen. We can now make it trace a line as it goes. Add these lines to your program.

```
 67 IF (ADVAL(0) AND 3)=1 THEN PROCDRAW

230 DEFPROCDRAW
240 GCOL 0,3
250 PLOT 69,X%,Y%
260 GCOL 3,3
270 ENDPROC
```

Whenever you press the fire button the procedure, PROCDRAW is called. This procedure just prints a small point at the current position of the cursor. By keeping your finger on the button a continuous line is traced out as you move the joystick.

When you try this program you'll find that the joystick is a bit over-sensitive and drawing accurate pictures is very difficult. This method of using the joystick is great for some applications but not really suitable for this.

Instead we can really make use of the difference of the BBC micro joysticks and use the distance that you move the joystick handle to control the speed of the cursor's movement. Substitute these lines into your program.

```
 10 MODE 1
 20 PROCSETUP
 50 REPEAT
 55   PROCPRINT
 60   X%=X%+((65520-ADVAL(1))-32760)/800
 70   Y%=Y%+(ADVAL(2)-32760)/800
 80   PROCPRINT
120   IF (ADVAL(0) AND 3)=1 THEN PROCDRA
W
140   UNTIL FALSE
150 END
160 DEFPROCSETUP
170 VDU 23,224,0,24,24,102,102,24,24,0
180 VDU 5
190 GCOL 3,3
200 DRAW 0,1023:DRAW 1279,1023
210 DRAW 1279,0:DRAW 0,0
220 X%=640:Y%=512
230 PROCPRINT
240 ENDPROC
250 DEFPROCPRINT
260 MOVE X%-16,Y%+16
270 PRINT CHR$(224)
280 ENDPROC
290 DEFPROCDRAW
300 GCOL 0,3
310 PLOT 69,X%,Y%
320 GCOL 3,3
330 ENDPROC
```

Now we are adding on to the current cursor co-ordinates a value dependent on the joystick's position. You'll find that the accurate control of the cursor is much easier now.

After the joysticks, one of the most frequently used peripherals for the BBC is some kind of special display. You BBC micro comes complete with no less than three different display outputs. The most often used of these — the TV output — you are already well acquainted with. However, you are probably aware by now of the limitations of a normal television as a display for the BBC micro. The problem is that domestic televisions simply can't give sufficiently good pictures with microcomputers. This doesn't show much in most cases, but just try reading any text displayed in the 80 column modes 0 and 3.

If you are going to use your micro in these modes a lot; if, for example you are going to use it for serious business programs that require the full 80 columns, you will need a better display. You need a monitor. These can come in two forms — composite or video monitors and RGB monitors. Both of these are catered for on the

BBC micro, and there are sockets giving suitable signals at the back of the machine.

You will find that a monitor helps more than just 80 column text. All displays are improved. A monitor puts an end, once and for all, to the fuzzy edges and washed out colour that a TV gives your carefully designed pictures.

The penalty that you have to pay for this improved picture is of course the comparatively high cost of a monitor. It can be as much as the computer itself. However as a monitor takes no learning how to use, can be connected and working in a jiffy, and frees your TV for the Saturday film, it is well worth considering.

Many BBC owners may also consider purchasing a printer. Like a monitor, a printer can be an expensive item, but proves its worth to most buyers. Again the BBC is well provided to take these. There are two sorts of printer interfaces around in any large quantities. These are known as Centronics, or parallel, and RS 232, or serial, printers.

Both types have the required socket on the BBC micro. The RS 232 socket is at the back of the machine and the Centronics socket is under the micro, labelled 'printer'. Connecting a printer is as easy as a monitor but there is a little more skill in using it. A detailed explanation is not really in place here as the dealer that sells you the printer should be able to tell all. However a few points are in order.

In the normal course of events a printer will not do anything. It has to be told by the micro to print. On the BBC micro this is done with a VDU command, VDU 2. This command will send all text that goes to the screen to the printer as well. VDU 3 will stop the flow again.

You can't draw with a printer in the same way as you can on the screen. Any attempt to do so will result in not a lot happening. However, if you purchase a fairly well known brand of printer you will probably find that you can also buy a piece of software known as a 'graphics screen dump'. This will take anything already on the screen and print out a rendition using the printer. The quality of the dumped picture may not be wonderful but that is as good as you'll get without resorting to far more expensive means. If you're very proud of a screenful of art that you have produced, there is of course nothing to stop you taking a photograph of it for posterity.

There are several other sockets in your BBC micro that we have not mentioned. The disk drive socket just about speaks for itself. It is to this socket that you connect your one or two floppy disk drives if you can afford them. If you find that you do write a lot of long programs, or if your programs use large amounts of data then you will find that a pair of disk drives are invaluable, well worth the expense.

The other sockets on your BBC are less obviously practical. The chances are that you may never use them. For a start there is the

'user port', next to the centronics printer socket underneath the machine. Despite its name, the user port is pretty difficulty to use. It is really intended for use with machine code programs to operate your own external equipment. We haven't dealt with machine code in this book as it is a subject ten times as vast as Basic and at least that much more complicated. There is some commercial equipment available that uses the user port but there is nothing to say here about that.

The other sockets on your BBC micro are similar in nature. The 1 MHz bus socket (next to the user port) is the main connector for all external equipment to be added to the BBC micro. A 'bus' in computer talk is the name for all the connections inside a computer that allow various chips to communicate with one another. Bringing these connections out to the outside world allows other pieces of equipment to be added into the circuit. Acorn produce an adaptor that connects to the BBC micro through the 1 MHz bus socket to allow you to receive the broadcast teletext information on your computer and to load programs included in the broadcasts directly into your computer.

Acorn has continued the traditional nomenclature of computer connectors with the so-called Tube connector. This is a special sort of bus to allow you to connect what amounts to a second micro onto your present BBC. This 'second processor' will take over from your present and only processor and run your programs. The present one will be left with only the screen, keyboard, sound, and data storage to deal with. The idea of this is that many hands make light work. Your super-BBC will work even faster!

There is a vast range of electronic bits and pieces that you can connect onto your BBC. We have only glimpsed a few of them here. As you come to use your BBC micro more, you will come to master the equipment that is provided and crave for more to widen your repertoire.

In this book we have just opened the door to programming your BBC micro. There is now a vast area of expertise in front of you, both knowledge of computing generally and of the BBC micro in particular. It is hard work to come to grips with a microcomputer, but the midnight oil is not burnt in vain. You will find that when the television channels can provide no entertainment on your TV screen, there is nothing more satisfying than putting your own work up there.

Appendix A

Throughout this book we have kept very much to 'the' book. That is, we have taken no short cuts, but spelt out everything to the full. This is all well and good as it means that you learn the correct way of doing things and your programs end up legible and easy to follow.

However there are occasions when crime pays! The BBC micro is limited by one severe factor — its memory, or rather the lack of it. When you are using the higher resolution modes (0, 1, and 2), so much memory is taken up for the display itself that there is comparatively little left for your program and data. We need a method to compact your programs into the smallest possible space so that we can cram them into the small amount of remaining memory. We also frequently want our programs to go that bit faster. In this section we are going to look at a few ways of doing both of these.

On the whole, saving space and increasing speed are very difficult to do at the same time. Most of these ideas are conflicting. However some are common.

The most obvious trick to save space is to repeat parts of the program as rarely as possible. If a few lines of a program are used frequently then put them in a procedure and call that single section of the program as often as needed.

Spaces can also be left out a lot of the time. This speeds up the program too. BBC Basic requires very few spaces. Each useless one lost is an extra piece of memory for some useful part of the program and saves the interpreter looking at a wasted character. Every little can help in extreme cases. Some spaces, however, must be kept in. There must be some way for the interpreter to tell which are the Basic keywords and which are variables. The table below will give you some ideas as to what you can and cannot do.

Right	Wrong
X=VALUE AND NUMBER	X=VALUE AND NUMBER
X=VALUE AND NUMBER	X=VALUE AND NUMBER
X=VALUE%ANDNUMBER%	
X=VALUE$ANDNUMBER$	

As a general rule, the micro knows where keywords end and variables begin, but it doesn't know how long a variable name is

unless you show it with a space, dollar sign, bracket, or whatever. Usually the error message, 'No such variable' will appear when you make a mistake over-compacting a program.

For speed you should first remove as many of the procedures and subroutines as possible. Repeat areas of the program when and where they are needed. If you have to call on a section of program use a procedure rather than a subroutine. These are called faster. Pass all the variables that are to be used in the procedure to it in the brackets, like this

Slow	Fast
10 X=25	10 X=25
20 Y=50	20 Y=50
30 PROCADD	30 PROCADD (X, Y)
40 END	40 END
50 DEFPROCADD	50 DEFPROCADD (A, B,)
60 RESULT=X+Y	60 RESULT=A+B
70 ENDPROC	70 ENDPROC

The difference in running time is, of course, very slight but if the procedure is called repeatedly throughout the program then these differences mount up.

All procedures should be placed at the start of the program, for the greatest speed, the most used ones first (with a suitable GOTO as the first line to start the whole thing running) because the micro always starts looking for procedures from the beginning of the program and so this way it will find them quicker. Data can similarly be placed there.

Another way to speed things up is to use integer variables wherever you can. We have already seen the dramatic difference that this can have on program execution speed. Cramming everything onto as few a line numbers as possible is also a good move for speed. This cuts out much of the micros dithering between lines. Needless to say reducing the number of line numbers reduces the space taken up by the program as well.

A very subtle method of speed increase is to be careful with your variable names. Use single letters first (this saves space too!) and when you have used one of each of these, start on two letter names. This speeds things up because the micro looks up the present value for a variable in a sort of table. This is arranged single letters first, then double and so on in alphabetical order. If you had five variables called A, B, C, D, and AAAAAAA, the last would take a lot longer to look up despite the fact that it is all A's.

Space and time can be saved by leaving things out! NEXT's do not have to be followed by the name of the loop variable. The micro knows which variable you are talking about. If you don't mention it then it doesn't have to check that you're right. When you have nested loops, you can save even more space with a comma, like this.

```
10FORI%=1TO10
20FORJ%=1TO5
30NEXT,
```

That line 30 is equivalent to saying

```
30NEXTJ%,I%
```

Which is the same as

```
30NEXTJ%
40NEXTI%
```

Notice how all the spaces have been left out here, and integer variables are used.

The frequent VDU commands can also be compacted. Rather than saying

```
10VDU19,1,4;0;
20VDU19,2,1;0;
30VDU5
40BDU23;11,0;0;0;
```

All these can be put onto one single line, like this:

```
10VDU19,1,4;0;19,2,1;0;5,23;11,0;0;0;
```

All these methods of saving time and space should be used with care. If you don't need them then don't use them. Don't expect any miracles either! The best method to produce both fast and efficient programs is to write them properly in the first place.

All of the methods mentioned here make a program harder to read and follow through. They destroy the structure of a program and ruin its friendly nature. It is usually best to write a program as we have done in the rest of the book initially, and then, if needed, make the necessary changes to trim it down or speed it up.

Appendix B

In this section we are going to see how to hang on to the programs that you slave to produce. You will soon be writing programs of your own that will reach hundreds of lines in length. All this is lost whenever you turn off your BBC micro. The only programs that can survive the loss of power are those stored in ROM by the manufacturer — the interpreter and operating system. All *your* programs are stored in RAM which needs to be continually supplied with power to retain its contents.

Of course you could always leave the computer switched on for ever . . .

Rather than clock up the world's largest electricity bill, we store our programs in a form that can do without power. This permanment form is a recording, either on cassette tape or floppy disk.

As most owners of BBC micros at least start off with only cassette recorder storage, that is what we are going to concentrate on. There are ten commands associated with cassette program storage. Using them is very simple. They are

LOAD	*CAT
SAVE	*EXEC
CHAIN	*SPOOL
	*MOTOR
	*LOAD
	*SAVE
	*TAPE

The first two in the first column are the most important. The first command, LOAD, is used to transfer a program on a cassette tape into the micro's memory. After the command, LOAD, there must follow the name of the program that you want to load from cassette. This is enclosed in inverted commas like this.

```
LOAD "name"
```

Only the program with that name will be loaded into the computer. Others that may be on the tape will be ignored.

If you just want to load the next program that the micro comes across on the tape then you can tell it not to worry about the recordings' names by missing that bit out

```
LOAD ""
```

After you press <RETURN>, the micro comes back with the message, 'Play then return'. If you have a remote control facility on your cassette recorder, the motor will now be switched on and the little light on the keyboard marked 'cassette motor' will come on. If your cassette recorder has a remote control socket it is a good idea to use it. Wind the tape to just before the program, press play on the cassette recorder and then <RETURN> on your micro. The message 'Searching' should come up to tell you that the micro understands and is looking for the program, it will respond with 'Found' and the program's name along with a number. If another name is found then this will be displayed but your micro will keep on searching for the one you want.

You program is recorded in a series of short sections called blocks. The number following the programs name is the number of the block that the micro is listening to at the time. This number will increase as the recording runs. The number increases rather strangely because it is what is known as a 'hexadecimal' number. This is a number to the base of sixteen rather than ten. Hexadecimal counts like this

1 2 3 4. . .8 9 A B C D E F 10 11 12. . .18 19 1A 1B 1C. . .

This needn't worry you, but you may wonder, otherwise, what is going on. Hopefully the whole program will load into the computer without any problem, and a beep will tell you that it is finished. If, however, the micro misses a bit, then the message, 'Data ?' or 'Block ?' will appear, maybe even 'Rewind tape', followed by 'Searching' again. Rewind the tape a bit and try again.

Your BBC micro is very patient and will allow any number of goes on a stubborn block. Eventually the whole program will be loaded and you can treat it as though you had just typed it in.

Of course you will want to record your programs onto cassette first. This is done using SAVE. The name that you want to call the program is also required. This must be ten letters long or less. You can't use spaces but the underline character (–) does quite well. You use the command, SAVE, in a similar way to LOAD.

```
SAVE "name"
```

The message this time is 'Record then return'. Following this and you'll see the name that you gave the program along with the block number increasing as each block is recorded. Take out the cassette when the beep signifies that the recording is finished. That is all there is to it.

Those are the only commands that you really have to know to operate your cassette recorder with your BBC micro. However a word or two about the other commands that we mentioned will not go amiss.

CHAIN is very similar to LOAD. Using it is just the same. You can give a name or not as you please and the messages are the same, but there is one difference. CHAIN will load a program into the computer just like LOAD, but once it is in, it will then automatically RUN the program. This is very useful for setting up 'chains' of programs. The last line in each program is the command, CHAIN, which then loads and runs the next program. Like this

```
 10 REM program one
    .
    .
    .
 90 CHAIN ""
100 END

 10 REM program two
    .
    .
    .
190 CHAIN ""
200 END
```

Many commercial programs, such as games, use CHAIN in this way to load and run all the various separate programs that make up the whole piece of software. Then you only have to touch your tape recorder once to load all the programs. The 'Welcome' tape that comes with your BBC micro uses CHAIN to do this.

All the other commands concerned with tape program storage are preceded with an asterisk (*). This is because they are used directly by the operating system. The most importrant of these is * CAT.

This command is not a reference to a feline pet but stands for 'CATalogue'. *CAT will give you a directory of all the programs that you have saved onto a cassette. Nothing is actually loaded or saved so the program that is in your micro already is unaffected. All that happens is a list of the names and lengths of your stored programs is displayed.

When this command is used, no message is displayed, but the recorder's motor is started and the micro will then read each program that it comes across to find its name. The same error messages are displayed if something goes wrong, and you can then take the same action as you would if you were loading a program.

One incidental use for *CAT is that it turns on the motor of your cassette recorder. This is useful if you want to, say, rewind a tape to the beginning. However a better method, and one that leaves the micro free for other tasks and can be used easily inside a program, is to use *MOTOR. This just switches the cassette motor on and off and displays the result on the keyboard LED. *MOTOR 1 will switch on the motor and *MOTOR 0 will switch it off. The motor is always switched off when you first switch on the micro and after you press <BREAK>.

The next two commands that we are going to look at are a bit strange, and need a bit of explanation. Your micro stores a program in its memory in the most efficient way that it can. Rather than storing a Basic keyword such as PRINT as five letters, each taking up one memory location, it stores these words as a single code called a token. If you are interested you will find a list of the tokens used for all of the keywords on page 483 of the User Guide.

When a program is saved on cassette the program is recorded onto the cassette tape in this way. That is to say the tokens are used for all the keywords. We can, however, store a program in long hand, as it were, without the keywords being tokenised.

To do this we use the command SPOOL. Apart from the way in which SPOOL stores a program on tape, it is used just like SAVE. A name is required and the messages produced are the same.

Of course spooled programs cannot be loaded back into the computer using LOAD. When load is used, the micro is expecting a tokenised program. Instead we use *EXEC. This command takes a untokenised program from the cassette recorder and treats it just as though it were coming from the keyboard (and from an extremely fast typist). *EXEC is used just like LOAD. The real benefit of these two commands, *SPOOL and *EXEC, is that they can be used to tack one program onto the end of another. Here's how.

We will assume that we have written a program called PROGONE and this has been saved quite normally on a cassette. We have just finished writing PROGTWO and we want to add this onto the end of PROGONE.

All we have to do is to SPOOL a version of PROGTWO onto another cassette (*SPOOL "PROGTWO"), LOAD PROGONE into the micro (LOAD "PROGONE") and then *EXEC the untokenised version of PROGTWO on top of PROGONE (*EXEC "PROGTWO"). Because we are using *EXEC, it is just as though we had typed in PROGTWO as corrections and additions to PROGONE. using this method does mean, however that any lines in PROGTWO that have the same number as those in PROGONE, replace them. You can get around this by RENUMBERing PROGTWO to start at a figure well above the end of PROGONE, before you *SPOOL it.

We have just three more commands left. Two of these look rather familiar. *LOAD and *SAVE are very similar to their counterparts without asterisks. These commands are used for loading and saving machine code programs. As we are not dealing with machine code in this book we need not worry ourselves with them.

However *LOAD can be used to check that a program has been saved properly. When you have saved a program rewind the tape to the beginning of the recording and type in this

```
*LOAD "name" 8000
```

Never mind what this does. All you have to do is to watch the screen. The usual loading messages appear. If the micro finds any error in the recording, and an error message appears, you should SAVE your program again. This procedure does not effect the program in the micro so you can still do this.

The only remaining command is *TAPE. Your BBC micro normally saves programs at a fair speed. The rate at which a program or data is recorded is measured in 'baud'. It doesn't matter what this unit of measurement really means, all you need to know is that your micro saves programs at 1200 baud. This speed is conveniently fast but can be a bit unreliable to LOAD back if the tape is of poor quality or the connecting leads are a bit dodgey. We can switch the speed to a super-reliable 300 baud using the command *TAPE3. To switch to 300 baud, type in *TAPE 3. To switch back you can type in *TAPE 12 or just *TAPE, as 1200 baud is the default value.

You must remember, however, that program saved at 1200 baud can only be loaded at 1200 baud. Similarly programs saved with *TAPE 3 must be loaded after a TAPE 3. Recordings made at 300 baud are useful for long term, or very important, saved programs or data. Of course a recording at 300 baud will take four times as long to SAVE and to LOAD.

In addition to these commands there are a few golden rules to follow when loading and saving any form of program or data with a cassette recorder.

Firstly, be careful! Once you have recorded over a program there is no way that you can ever get it back. Use the break-off record-protect tab on the cassettes with valuable programs. Similarly once you have loaded a program into your micro and overwritten one that is already there you can never retrieve it.

Because of this it is always a good idea to make back up copies of all your programs. Always make two recordings of a finished program and check them with the *LOAD method immediately.

As you develop a program on the computer it is wise to make a

recording of what you have done so far, just in case.

Keep a record of which programs you have on which cassettes, and where they are on that cassette. If your recorder has a tape counter this is very easy, just get into the habit of always zeroing the the counter at the beginning of the tape and make a note of the count when you start to save a program.

If you stick to these rules, you'll have hundreds of hours of trouble-free cassette use. Floppy disk drives will always be a better system of program and data storage than cassettes, but for the money you will find that cassettes are pretty good too.

Appendix C

Many of the features of the BBC micro can be controlled with suitable *FX commands. They are used to set up numerous background aspects of the BBC's operation. Aspects that you may take for granted and assume cannot be changed.

Some of the *FX commands are detailed in the User Guide, however that is by no means a complete list. Not all of the *FX commands have yet been allocated. Some are kept for future expansions to the BBC micro system. The list here is not complete either, but it will give you an idea of the commands most useful to you.

No really detailed explanation is given for any of these commands. You may be able to find more information in the User Guide for some codes. If you can't, you can easily experiment with them and find out for yourself. Most of them you will never use. Many of them relate to external pieces of equipment, like printers. There are however many really useful ones and it seems unfair to deem which you should be told and which to withold.

*FX 0 Prints Operating system version number.
This book has assumed that you have version 1.2. Check it.

*FX 1 For user machine code routines. Complicated!

*FX 2,n select input source:
- 1 keyboard
- 2 RS 423 serial port
- 3 Keyboard and RS 423
- default = 0

*FX 3 select output destination
- 0,1,4, or 5 screen
- 0,1,2, or 3 printer
- 1,3,5, or 7 RS 423
- 6 none
- default = 0

*FX 4,n select use of cursor keys
- 0 editing
- 1 ASCII codes
- 2 User definable keys
- default = 0

*FX 5,n select printer destination
- 1 centronics port
- 2 RS 423

*FX 6,n — set character ignored by printer. n=ASCII code

*FX 7,n — set RS 423 receive baud rate. n=rate.

*FX 8,n — set RS 423 transmit baud rate.

*FX 9,n — set duration of first colour of flashing pair in 1/50th sec
0 gives steady first colour
default = 25

*FX 10,n — set duration of second colour of flashing pair in 1/50th sec
0 gives steady second colour
default = 25

*FX 11,n — set key auto repeat delay in 1/100th sec
0 gives no auto repeat
default = 32

*FX 12,n — set key auto repeat interval in 1/100th sec
0 gives default values for delay and interval
default = 8

*FX 13,n — disable interrupt events

*FX 14, n — enable interrupt events

*FX 15, n — flush internal buffers
0 all buffers
1 current input buffer flushed

*FX 16,n — set number of ADC channels sampled. Low numbers gives faster sampling
default = 4

*FX 17,n — force conversion on ADC channel n

*FX 18 — reset user definable keys to contain nothing

*FX 19 — wait for next frame of TV display

*FX 20,n — explode character RAM allocation. n gives character block.

*FX 21,n — flush specific buffer
0 keyboard buffer
1 RS423 input
2 RS423 output
3 printer
4 sound channel 0
5 sound channel 1
6 sound channel 2
7 sound channel 3
8 speech buffer

*FX 111,n — switch between data areas
data area zero
data area one

*FX 119 close any SPOOL or EXEC files
*FX 124 clears <ESCAPE> condition
*FX 125 simulate <ESCAPE> key being pressed
*FX 126 attempt to clear <ESCAPE> condition
*FX 136 execute user routine at USERV
*FX 137,n switch cassette motor. Equivalent to *MOTOR
0 motor off
1 motor on
default = 0
*FX 138 insert value into buffer
*FX 139 select file options. Equivalent to *OPT
*FX 140 select filing system. Equivalent to *TAPE
*FX 141 select ROM filing system
*FX 142,n enter language ROM nth from right
*FX 143 issue service ROM request
*FX 144,n,m, alter display parameters
n=1 display one line up on mode change
n=255 display one line down on mode change
m=0 interlace on
m=1 interlace off
*FX 146 to 151 read/write to mapped I/O
*FX 153n,m insert character into input buffer
n=0keyboard buffer — simulate keyboard entry
m=character code
*FX 154 to 155 write to registers of video ULA
*FX 156 read update video control chip register
*FX 157 equivalent to BPUT
*FX 159 write to speech processor if fitted
*FX 181 read/write RS423 mode
*FX 183 select tape/ROM filing system
*FX 194 equivalent to *FX 9
*FX 195 equivalent to *FX 10
*FX 196 equivalent to *FX 11
*FX 197 equivalent to *FX 12
*FX 200,n select <ESCAPE>,<BREAK> effect
0 or 2 normal <ESCAPE> action
0 or 1 normal <BREAK> action
2 or 3 memory cleared on <BREAK>
1 or 3 <ESCAPE> disabled
default = 0

*FX 201	enable/disable keyboard 0 enabled 1 disabled
*FX 202	read/write keyboard status (caps lock, <CTRL>, etc)
*FX 203	read/write RS423 handshake
*FX 205	select tape/RS423 0 RS423 data to RS423 64RS423 data to cassette hardware
*FX 206 to 208	econet
*FX 210	select sound 0 disabled 1 enabled
*FX 212,n	sound channel number for 'bell' (CHR$(7)) default = 3
*FX 213,n	gives bell frequency n default = 101
*FX 214,n	gives bell duration default = 7
*FX 219,n	define <TAB> key character to code n default = 9
*FX 220,n	ASCII character to give ESCAPE 65 gives A as <ESCAPE> 0 disables ESCAPE default = 27
*FX 234	alter flag giving TUBE presence
*FX 235	alter flag giving Speech processor presence
*FX 236	read output destination as set by *FX3
*FX 237	read edit key status as set by *FX4
*FX 252,n	sets ROM number to be entered after break
*FX 254	read/write available RAM 0 gives 16k after <CTRL><BREAK> 255 gives 32k after <CTRL><BREAK>
*FX 255,n	set start up options following <CTRL><BREAK> 3 gives mode 3 7 gives mode 7 and so on

THE BETTER GUIDE TO THE BBC MICRO

Everything you want to know about the BBC Micro that the other computer books and the user's manual don't tell you!

The Better Guide to the BBC Micro is the result of months of research by leading home computer experts.

An essential guide, it starts from scratch for the absolute beginner and goes beyond the "official" user's manual, exploring the greater potential of the BBC Micro home computer.

Covering in detail questions not usually dealt with elsewhere and offering many "tricks of the trade", this is the definitive reference book for those who want to get the most out of their home computer.

A genuinely practical guide to the BBC Micro

ISBN 0-946391-52-1

00395

9 780946 391523

ZOMBA BOOKS

0 946391 52 1

£3.95